INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

OPTIMIX Language Manual

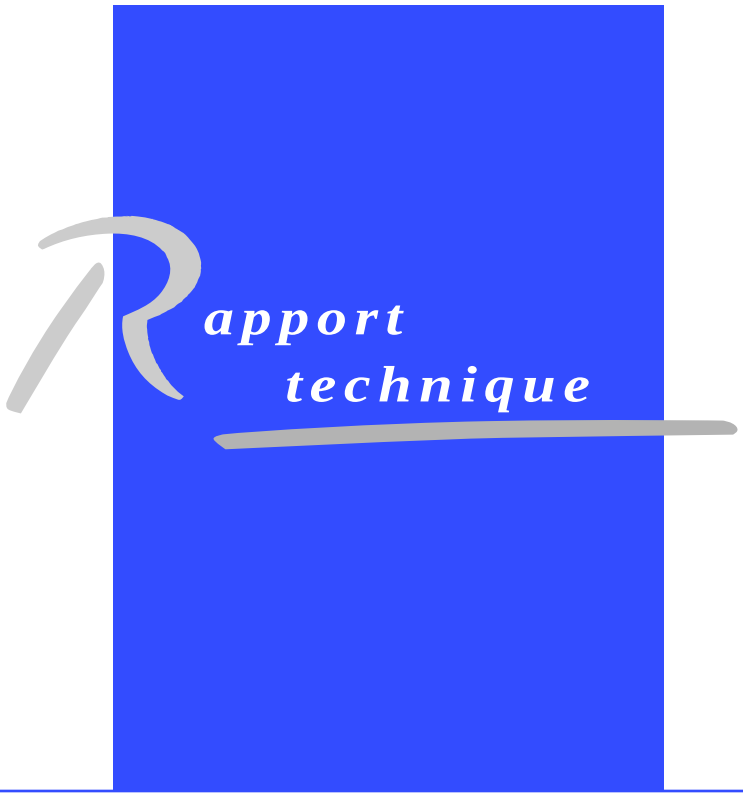
(for OPTIMIX 2.0)

Uwe Aßmann

N° 0195

August 1996

THÈME 2



ISSN 0249-0803

OPTIMIX Language Manual

(for OPTIMIX 2.0)

Uwe Aßmann*

Thème 2 — Génie logiciel
et calcul symbolique
Projet OSCAR

Rapport technique no0195 — August 1996 — 54 pages

Abstract: This is the language manual for OPTIMIX, the optimizer generator. OPTIMIX can be used to generate program analyses and transformations. Its input language is based on DATALOG and graph rewriting. Especially two new classes of graph rewrite systems are used: edge addition rewrite systems (EARS) and exhaustive graph rewrite systems (XGRS).

Key-words: Compiler generator, program optimization, specification, stratification, very high-level languages, visual programming

(Résumé : tsvp)

I would like to thank Martin Jourdan and my other colleagues from the group OSCAR at INRIA Rocquencourt. During my post-doc year they provided an excellent environment to write this paper. This work has partly been supported by Esprit project No. 5399 COMPARE.

* Uwe.Assmann@inria.fr. From Nov. 96 on: University of Karlsruhe, Institut für Programm- und Datenstrukturen, Vincenz-Priessnitz-Str. 3, 76128 Karlsruhe, Germany, assmann@informatik.uni-karlsruhe.de

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Manuel du langage OPTIMIX

(pour OPTIMIX 2.0)

Résumé : Ceci est le manuel décrivant le langage d'OPTIMIX, le générateur d'optimiseurs. OPTIMIX peut être utilisé pour générer des analyses et transformations de programmes. Son langage d'entrée est basé sur DATALOG et la réécriture de graphes. Deux nouvelles classes de systèmes de réécriture de graphes sont particulièrement utilisées : les systèmes d'ajout d'arcs (*edge addition rewrite systems*, EARS) et les systèmes de réécriture de graphe exhaustifs (*exhaustive graph rewrite systems*, XGRS).

Mots-clé : Générateur de compilateurs, optimisation de programmes, spécification, stratification, langages de haut niveau, programmation visuelle

... there clearly remains more work to be done in the following areas:
(1) discovery of other properties of transformations that appear to have relevance to code optimization,
(2) development of simple tests of these properties, and
(3) the use of these properties to construct efficient and effective optimization algorithms that apply the transformations involved.

Aho, Sethi, Ullman in "'Code Optimization and Finite Church-Rosser Systems"' [ASU72]

22. Der Einsatz von Graph-Ersetzungssystemen auf Probleme der Datenfluß-Analyse, wie Bestimmung gemeinsamer Teilausdrücke, Parallelisierbarkeit, Schleifenoptimierung, existiert bisher nur in Andeutungen... Auch das Problem der Übersetzung eines Zwischencodes (in Form eines Programmgraphen) in Maschinencode mit Hilfe von Graph-Ersetzungssystemen wurde bisher kaum untersucht.

M. Nagl in [Nag79], Kapitel Offene Probleme

Contents

Warning

This is a language reference manual, not a tutorial on the OPTIMIX-specification language.

OPTIMIX is, although it provides quite advanced rule-based programming concepts for specification of optimizations, not a commercial software product, but a *research prototype*. Thus

1. OPTIMIX may dump core in unexpected situations. Then send a mail with the specification and the version number of your binary to the author. To obtain the version number call `optimix -v` on the command line. I do not promise to provide immediate support but I am glad if users help me to detect errors.

2. OPTIMIX may generate incorrect code. Test your generated code first before you trust it!!

3. The documentation may be too crude to understand (because I have only limited time for it). Please try OPTIMIX on different variants of your specification. Also read all papers on the subject of graph rewriting for program optimization.

4. The specification language is not stable, and may change in further versions. The syntax is sometimes inconsistent and non-orthogonal (large and small keywords, bracket structuring). This is due to historic reasons and the experimental nature of the language. Future versions will try to make the language more orthogonal and in its conventional pars more similar to classical languages as C/C++.

5. OPTIMIX is compiled with an enourmous amount of gcc warnings. This is due to the use of void pointer classes and the AST tool types, which would need casting everywhere. If the system does not compile, send a mail to the author and use the OPTIMIX-binary.

6. To construct OPTIMIX has cost me about 3 man years. Please be patient with me and the current status of the tool.

7. If you are pleased with the tool and find it revolutionary, please send a plain mail postcard with a nice picture to my address. This will motivate me to fight for further development. The more encouraging comments are found on the postcard, the better!

Chapter 1

General Topics

This is the language manual for OPTIMIX, the optimizer generator. OPTIMIX can be used to generate program analyses and transformations in C language. Its input language is based on DATALOG and graph rewriting [Aß94] [Aß95] [Aß96b]. Especially two new classes of graph rewrite systems are used: edge addition rewrite systems (EARS) and exhaustive graph rewrite systems (XGRS).

The development of OPTIMIX has partially been supported by the Esprit project COMPARE (No. 5399). The tool is not in the public domain; however, a free version can be ordered from the author.

It is highly recommended that the user first reads the papers [Aß94] [Aß95] [Aß96b] [Aß96a]. [Aß94] and [Aß96a] are available with the OPTIMIX-package.

1.1 Design procedure for an optimizer

In order to generate optimizer parts with OPTIMIX we propose the following procedure [Aß96b].

1. Write down all preconditions for a transformation, perhaps in text.
2. Define the data model of your application, i.e. define which parts of the knowledge you want to present should be objects and which should be relations. This can be done either with the syntax of the tool AST from the GMD toolbox Cocktail [GE90] [Gro89], or — within the context of the compiler model CoSy — in CoSy-fSDL [Buh95].
3. Design of the data manipulation, i.e. formulate graph rewrite systems that compute and transform the graphs that were defined in the data model. Build graphs with edge addition rewrite systems (EARS), and transform them via general graph rewrite systems (GRS).
4. Think about the representations of the graphs. Which algorithms does OPTIMIX generate for a problem and with which graph representations do these run fast? Exchange graph representations (functor calls) accordingly.

1.2 Two principal modes

OPTIMIX can be used in CoSy-fSDL-mode and in AST/standalone-mode.

CoSy-fSDL-mode This mode is only feasible in the CoSy compiler construction environment. The data model is specified in CoSy-fSDL, as common files and views of engines, that collaborate in the compiler. For each compiler, CoSy produces a *flatform-file* which contains all fSDL specifications in a flat form. OPTIMIX must read this file to know about the data model (option `-ff`).

AST/standalone-mode OPTIMIX is able to read existing data model specifications of AST (with small restrictions) [GE90] [Gro89]. This means that existing AST data specifications of compilers can be reused and extended for use with OPTIMIX. AST data definitions are module-based and AST-modules may occur within OPTIMIX-specification files. Thus AST data definition language can also be used as standalone data definition language of OPTIMIX.

The compiler construction toolbox Cocktail is available in two forms. There is a free version at `ftp://i44ftp.info.uni-karlsruhe.de/pub/cocktail/`. After 1993, Josef Grosch, the developer of Cocktail, has become independent and maintains Cocktail within his own company. Currently there is an improved version of the toolbox, e.g. containing a new LR(k) parser generator with visual debugging aid. A research licence is available for this version.

```
Dr. Josef Grosch
CoCoLab
Hagsfelder Allee 16
D-76131 Karlsruhe
Germany

Tel.: +49-721-697061
Fax : +49-721-661966
Mail: grosch@cocolab.sub.com
```

1.3 A first example

First consider a small program analysis example, the transitive closure over basic blocks. Typically, basic blocks are sequences of statements started by labels and ended by jumps (straight-line code). Each block is connected to other blocks via the jumps and the labels. A block `B1` is predecessor of another block `B2` if it ends with a jump to `B2`. Then also `B2` is a successor to `B1`, and this relation makes up the basic block graph.

If we want to know which blocks are reachable from a block, we have to construct the transitive closure operation on the basic block graph. Assume our block looks like (in AST):[1]

```
MODULE TransitiveClosureDDL TREE MyTree RULES
Block =
   /* other attributes left out */
   /* the successors in the basic block graph as embedded neighbor */
   /* sets (type consset(Block)) */
   ( BlockGraph: consset(Block) )
   /* the successors in the reachable block graph as embedded */
   /* neighbor sets (type consset(Block)) */
   ( ReachableBlocks: consset(Block) )
.
END TranstiveClosureDDL
```

A similar CoSy-fSDL definition would be:

```
domain Block { Block <
   /* the successors in the basic block graph as SET functor application */
   BlockGraph:      SET(Block),
   /* the successors in the reachable block graph as SET functor application */
   ReachableBlocks: SET(Block)
>};
```

With this data model we may write a specification that computes the transitive closure of the graph `BlockGraph` into the graph `ReachableBlocks`:

[1] Actually this is an extension of AST syntax. See section 2.3.3 how OPTIMIX collaborates with AST.

```
MODULE TransitiveClosure
OPT
EARS ComputeReachableBlocks(BlockSet:consset(Block))
{
   RANGE b <= BlockSet;
   RULES

   ReachableBlocks(b,b1) :- BlockGraph(b,b1).
   ReachableBlocks(b,b1) :- BlockGraph(b,s), ReachableBlocks(s,b1).
}
END TranstiveClosure
```

This EARS (edge addition rewrite system) specifies with two rules how a relation `ReachableBlocks` over blocks may be constructed by querying another relation `BlockGraph`. The first rule means that a block `b1` which is a successor to a block `b` in relation `BlockGraph` should also be a successor in Relation `ReachableBlocks`. The second rule describes the transitive closure: if there is a successor block `s` to `b` in relation `BlockGraph` which has another reachable block `b1`, then `b1` should also reachable from `b`.

For the EARS a C routine with name `ComputeReachableBlocks` is generated. This routine walks over all blocks from the parameter set `BlockSet` and applies the two rules. Because the rules are recursive, the rule applications are embedded in a fixpoint evaluation loop. To see which code OPTIMIX generates for this specification, feed the following file (`example-reachable.ox` from directory `doc`) to OPTIMIX. The generated C code may be found in Appendix 6.3.

```
//////////////////////////////////////////
// Transitive closure over basic block graph.
//////////////////////////////////////////

MODULE TransitiveClosureDDL
TREE MyTree RULE
Block =
   /* other attributes left out */
   /* the basic block graph as embedded neighbor sets (type set_Block) */
   ( BlockGraph: consset(Block) )
   /* the reachable block graph as embedded neighbor sets (type set_Block) */
   ( ReachableBlocks: consset(Block) )
.
END TranstiveClosureDDL

MODULE TransitiveClosure
OPT
EARS ComputeReachableBlocks(BlockSet:consset(Block))
{
   RANGE b <= BlockSet;
   RULES

   ReachableBlocks(b,b1) :- BlockGraph(b,b1).
   ReachableBlocks(b,b1) :- BlockGraph(b,s), ReachableBlocks(s,b1).
}
END TranstiveClosure
```

1.4 Running OPTIMIX from shell

```
shell>> optimix [options] filenames
shell>> optimix [options] < filename
shell>> optimix -typedefs [options] filenames
```

OPTIMIX can be run as standalone command (line 1), or as a filter in a pipe (line 2). Thus a previous run of `cpp` can be used to resolve any conditional #ifdef-commands in a specification. If the special option `-typedefs` is set, only pointer type definitions for the used C types are produced (line 3).

If the user specifies several OPTIMIX input files, the generated files are prefixed by the file name prefix of the first file. For instance,

```
optimix optimizer.ox optimizer2.ox
```

will create two files `optimizer.c` and `optimizer.h` which contain the generated code and its interface. Also the option `-o <file>` overrides the name of the output file to `<file>`.

File-suffixes need not be specified. Then OPTIMIX looks for the file in the current directory, trying the following suffixes in order: `.ox`., `.cg`, `.ast`.

OPTIMIX can be parametrized in different ways. There are options, which are set in the specification file, and command line options. Command line options fall in directly recognized options (e.g. `-typedefs`) or option keywords after the generic prefix option `-x` (e.g. `-x PrintSuperClasses)` The latter ones refer to options which are not often used or development options. It is likely that in future version of OPTIMIX option keywords become directly recognized options. The command line options of OPTIMIX are:

	— General Options: —
-help (-h)	print this message and exit
-x *name*	set option flag *name*
-x *name=value*	set option keyword *name* to *value*
	— Input Options: —
-ff *name*	use name for fSDL flat form file; CoSy-fSDL mode is turned on
-ast *name*	use name as AST/CG data specification file
	— Output Options: —
-o *name*	use name for output files
-typedefs	if AST data specifications are used, only print a file `oxtypes.h` with forward declarations for C types of flat sol functor applications.
-x InternalTypedefs	if AST data specifications are used, print all forward declarations for C types of flat sol functor applications in the .h-file. Otherwise they are printed in the external file `oxtypes.h`
-comments *value*	(currently has limited effect) emit generated code with comment level *value*. The larger *value*, the more comment is printed. 3 is current maximum.
-SimpleNestedLoopJoin	use nested loop join code generation (default code generation mode)
-ETJoin	use element-test-join code generation
-ETFilters	use element-test path filters during code generation
-nobitsetopt	do not generate bitset operations in code
-noindex	do not use index optimization
-view *name*	(CoSy-fSDL mode) use name as view name of the engine
-helpfuns	(CoSy-fSDL mode) produce help functions together with other functions
-helpfun *name*	(CoSy-fSDL mode) produce help functions in file *name*
	— Verbosity: —
-v	print version info, do nothing else
-VersionNumber	print version number only, do nothing else
-v1	be a bit verbose (default)
-v2	be fully verbose
-silent	be totally silent
-poem	print a poem and exit
	— Debug information: —
-x ShowSuperClasses	print the super classes of all class or fSDL domains.
-x ShowClassAttributes	print the classes and their attributes (only for AST classes)
-ShowOptions	print all options which are set
-ShowBinding	print bindings of types for variables during type inference
-ShowComparing	print comparisons of types for variables during type inference
-ShowSignatures	print signatures of all rules (types of rule test root nodes).
-ShowNodeTypes	print all infered types of all variables.
-ShowRTG	write all rule test graph in VCG format to files
-ShowRTGPaths	print all paths of path coverings in rule test graphs.
-ShowTermination	print the termination labels of each rule and stratum
-prio *int*	print test outputs that have priority less than *int*
-diag *name*	(obsolet) use diagnostic output file *name*
-ShowParseTree	(obsolet) write internal data structures of ox in ASCII format
-parser	run only the parser

1.4.1 Option file

Instead of giving the command options on the command line, the user can pass them also via a customization file, `.optimixrc`, which must be located either in the current directory or in the home directory. Each option (maybe also with a value) has to stand on an extra line in the file. Empty lines and lines beginning with `#` are ignored.

Chapter 2

An Optimix Specification

2.1 Outline

The outline of an OPTIMIX-specification file is the following:[1]

```
OptimizerSpecification ::= 'MODULE' ModuleName 'OPT' [ OptimizerName ] GlobalTargetCodeSections
   [ fSDLImportDeclaration ] [ InheritanceDeclarations ] GraphRewriteSystems 'END' ModuleName
   | 'OPT' [ OptimizerName ] GlobalTargetCodeSections
   [ fSDLImportDeclaration ] [ InheritanceDeclarations ] GraphRewriteSystems .
GlobalTargetCodeSections ::= [ 'HFIRST' TargetCodeBlock ] [ 'IMPORT' TargetCodeBlock ]
   [ 'EXPORT' TargetCodeBlock ] [ 'GLOBAL' TargetCodeBlock ]
   [ 'BEGIN' TargetCodeBlock ] [ 'CLOSE' TargetCodeBlock ] .
GraphRewriteSystems ::=  ( UseClause | GraphRewriteSystem ) * .
```

OPTIMIX-specifications are module-based. OPTIMIX accepts OPTIMIX-modules and AST-modules.[2] As in AST, OPTIMIX-modules are bracketed by the pair `MODULE <module-name>` and `END <module-name>`. However, in order to distinguish OPTIMIX-modules from AST-modules, the keyword `OPT` is used instead of the AST keyword `TREE`. `OPT` may be followed by an `OptimizerName`, which then is the name of the output file.[3] If the specification is monolithic, the module bracket may be left out. Then the file is regarded as one module which does not contain AST-modules.

Specified modules are combined in order to compose the output file(s). The graph rewrite systems of all OPTIMIX-modules are collected into one list. The code which is generated for them is put together into one C output file. In AST/standalone-mode, OPTIMIX also collects all data specifications from AST-modules, coalesces them into a single data model, and type-checks the graph rewrite specifications against this model. In CoSy-fSDL mode, the user must specify a flatform-file from which the flat fSDL data specification can be read.

As AST, OPTIMIX accepts several global target code sections, containing code of the target language C. The code is copied unchanged to certain parts of the generated files:

HFIRST into `<OptimizerName>`.h file; before any code line. Can be used to manipulate inclusions of files

IMPORT into `<OptimizerName>`.h file; after the inclusion of stdio.h

EXPORT into `<OptimizerName>`.h file; after IMPORT

GLOBAL into `<OptimizerName>`.c file; after the prologue

[1] Note that the grammar parts we give here are not the actual grammar of the parser; they only show the layout of an OPTIMIX specification.

[2] AST-modules are described in section 2.3.3.

[3] Otherwise it is by default the first command line file argument or the value of the `-o` option.

BEGIN into `<OptimizerName>.c` file into the begin function `<OptimizerName>_Begin()`

CLOSE into `OptimizerName.c` file into the close function `<OptimizerName>_Close()`

2.2 Lexical parts

Lexical items of OPTIMIX specifications are the following:

```
String ::= ''' any ''' |  '"' any '"'.
Digit ::= [0-9] .
Integer ::= Digit + .
Ident ::= A-z (A-z|Digit)+ .
TargetCodeBlock ::= '{' any '}'.
TargetPredicate ::= '{*' any '*}'.
```

Keywords Special keywords are the following.[4] Also their counterparts in lowercase letters are reserved, denoting the same.

```
ADD       AFTER     ANY
BEGIN     BEFORE
CHECK     CLOSE     CONSLIST  CUT
DAG       DECLARE   DELAYEDREMOVE       DELETE
EARS      END       ENDINPUT  EXPORT    EXPORTS
FAIL      FALSE     FINER     FIRST     FIRSTFIX FOR     FORALL
FREE      FUN       FUNCTION  FUNCTOR
GLOBAL    GENERIC   GRAPH     GRS
HASH      HFIRST    HYPEREDGE
IMPORT    IMPORTSDL INDEX     INHERITED INITIAL  INPUT   ITERATE
LAST      LASTFIX   LEFT
MARK      MODULE
NEW       NEXT      NIL       NOT       NULL
OPT       OPTIONS
PATH      PRED      PREV      PROC      PROPERTY
RANGE     REDUCIBLE REMOVE    REFINE    REUSE    RULE    RULES
STRATUM   SUCC
TARGET    THREADED  TREE      TRUE
USE
VIEW      VIRTUAL
XGRS
```

Note that within AST-modules AST-syntax holds. Because for AST code the same parser is used, all keywords of AST are reserved also within OPTIMIX-specifications. Also the following are special keywords of OPTIMIX. They are names of functors (template classes, either from CoSy-fSDL or the sol-library, see section 2.3.1):

```
LIST      SET       SETF
EGRAPH    SGRAPH    HGRAPH    SEQCLASS
BIPUNI    BITUNI
bitset    conslist  consset   hashset   ptrarray
bipuni    bituni    hgraph    seqclass
```

Delimiters Delimiter of identifiers (besides white space (space, newline, tab)) are:

```
( ) { } { } . ; : :- <-> //  /* */ (* *) {* *} {| |} {|| ||}
{# #} (| |) -> ~ !~ == < > ! ? :=
<= ==>
```

[4] Some of them are not yet used

Line comments are started by `//` and end at a newline. Non-nested comments start with `/*` and end with `*/` (same as in C++). There are also nested comments available as in Modula: `(* nested comment *)`. It is not allowed to use the string delimiter characters `'` and `"` in comments. The keyword `ENDINPUT` ends the input in a specification file, i.e. all text after it is regarded to be a comment. This is nice for testing; just move text after `[ END <module-name> ] ENDINPUT` and OPTIMIX will not see it.

Basic syntactical definitions We will need some syntactical definitions in the following:

```
Type ::= Ident .
C-Type ::= FlatFormType | Ident .
Variable ::= Ident .
GraphName ::= Ident .
Name ::= Ident | String .
fSDLDomain ::= Ident .
fSDLOperator ::= Ident .
FieldName ::= Ident .
FlatFormType ::= Ident .
ActualParameter ::= Ident .
StringList ::= String // ' '  .
IdentKommaList ::= Ident // ' '  .
NodeType ::= fSDLDomain '@' fSDLOperator
   | fSDLDomain | fSDLOperator | Type .
```

A C-Type is a type which can be understood by the C-Compiler. A FlatFormType is a C-type which results from functor flattening in fSDL, i.e. from expanding all inheritance relations and functor calls.

2.3 Global data declarations

OPTIMIX requires that the user specifies a *data model* of the graphs which are queried and rewritten (data definition specifications). The data model is specified in a *data definition language (DDL)*, either AST-DDL or flat-CoSy-fSDL.

2.3.1 Predicates, object types and functors

Graph representations by graph functors

In OPTIMIX analysis and transformation specifications use rules that are conjunctions of binary predicates (sections 3.1 – 3.7). This is the style of DATALOG [CGT89b]. Each predicate is binary and must correspond to an edge of a graph of the data model. The data model specification tells how these graphs, i.e. the predicates, are represented. OPTIMIX provides *graph representation transparency (functor transparency)*: it is transparent from a predicate specification how a corresponding graph is represented, this is only expressed by the type of the object field in the data model.

Types of graphs and sets are expressed by *functor calls* on object types. Functors are template classes, which are instantiated by one or several object types in order to specify a concrete graph or set. The given predicate name of the specification is used to find the object field in the data model, and with that the functor call. The code to traverse the concrete graphs is generated according to the functor call. A change of the type of a graph or set field, i.e. a change of a functor call changes the generated code, while the rewrite specification stays the same.

If graphs are implemented with these functors, you can test whether certain edges exist, and add or delete edges from them. OPTIMIX also understands simple pointer fields. You are allowed to navigate via them by writing down their field name as predicate.

Typechecking

A rewrite specification in an EARS or GRS is checked against the data model in the following way (Figure 2.1).

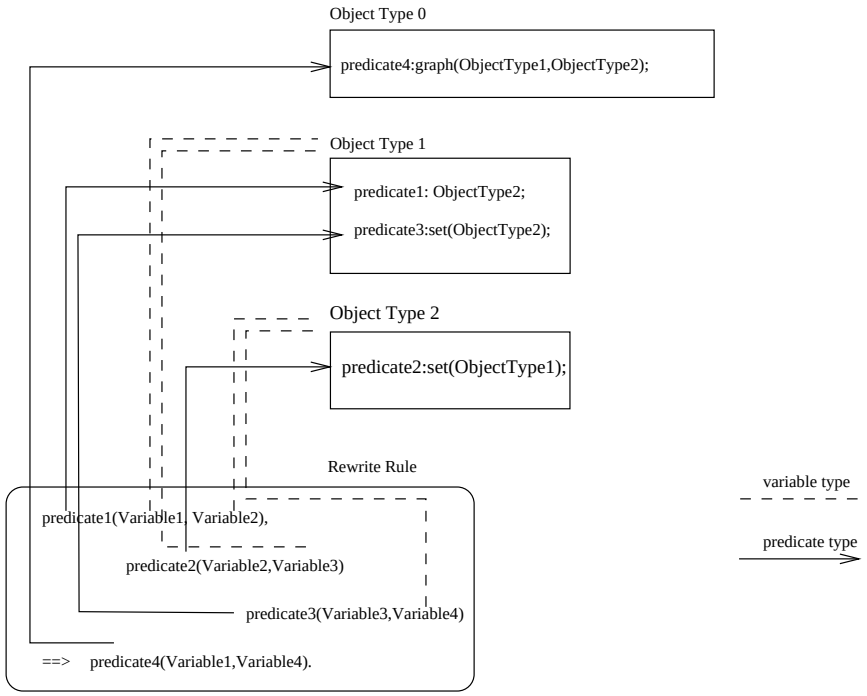


Figure 2.1: Typechecking a rewrite rule against the data model. Predicates refer to field names (solid lines), variables to object types (dashed lines).

1. for all predicates, lookup the predicate as a field in an object.
2. determine the type of the field.

   object type (pointer type) (in the figure `predicate1`) Then the relation is one-valued. The containing object type of the predicate determines the type of the left variable of the predicate. The right type is the type of the field.

   set functor type (in the figure `predicate2`) Then the relation is multi-valued, and represented by a call to an embedded set functor. This means that the field consists of a set of neighbors of an object type (the parameter of the functor call). This parameter determines the type of the right variable of the predicate. The containing object type of the predicate determines the type of the left variable of the predicate.

   list functor type Same as previous case, except that the neighbor set is ordered.

   graph functor type (in the figure `predicate4`) Then the relation is a functor-created graph over two types. The first parameter makes up the type of the left variable, the second that of the right variable.

3. When all types of variables for predicates have been determined, check whether these are compatible (equal or subtypes of each other).

OPTIMIX uses the type information on variables and predicates to generate correct navigation and manipulation code. This code calls functions from the functor libraries (loop over neighbor sets, add

edges, delete edges, delete nodes, add nodes, test existance of edges). The variable type information is also used to check whether the user has specified correct pattern matching.

Currently OPTIMIX only supports a fixed set of functors, either from the CoSy functor library or from the sol-library which is shipped in the distribution.

2.3.2 Data definitions with CoSy-fSDL

Within the CoSy compiler environment OPTIMIX can be used to produce engines which are put into CoSy compilers.

Available functors for CoSy-fSDL

Set functors

```
Functor ::= fSDLFunctor | SolFunctor  .
fSDLFunctor ::= fSDLHomogeneousGraphFunctor | fSDLBipartiteGraphFunctor | fSDLSetFunctor .
SetFunctor ::= 'SET' | 'LIST' | 'SETF'.
FunctorCall ::= Functor '(' NodeType // ',' ')'.
```

Types of set fields may be instantiated via set functor calls. These consist of an application of a functor (a template class) to one or several node type parameters. Please consider the CoSy framework documentation for set functor parameters.

Graph functors OPTIMIX supports *functor-created* as well as *hand-crafted* graphs. Functor-created graphs provide a set of nodes as well as the relation. Hand-crafted graphs always consist only of the relations, which are represented as neighbor sets in objects. Thus hand-crafted graphs are always represented as sets of neighbor sets, and their node set is not represented explicitly. In CoSy-fSDL-mode the supported functors are:

```
fSDLFunctor ::= fSDLHomogeneousGraphFunctor | fSDLBipartiteGraphFunctor | fSDLSetFunctor .
fSDLHomogeneousGraphFunctor ::= 'EGRAPH' | 'SGRAPH' | 'HGRAPH' | 'SEQCLASS'.
fSDLBipartiteGraphFunctor ::= 'BIPUNI' |  'BITUNI' | 'SETFUNI'.
```

Predicates in rules may also refer to fields with object types (pointer types). Navigation is done also by specifying their field name as predicate in a rule. OPTIMIX generates the corresponding field dereferencing.

Import a flat form file

Every CoSy compiler relies on several CoSy-fSDL data specifications, and especially on an CoSy-fSDL flatform-file `<compiler>.fdl`. This flatform-file must be imported to every OPTIMIX-specification. The user can import a flatform-file via command line option `-ff` (section 1.4). If the specification is used only for one compiler, the import can also be specified in the OPTIMIX-specification itself:

```
fSDLImportDeclaration ::= 'IMPORTSDL'  String  .
```

This declares that OPTIMIX should read an CoSy-fSDL flat form file with name `String` and turns on CoSy-fSDL-mode.

Inheritance declarations for CoSy-fSDL

```
InheritanceDeclarations ::= 'FINER' FinerDecl * .
FinerDecl ::= Ident // '<' ';'.
```

The CoSy-fSDL flatform does not contain inheritance information because domains are flattened. Because OPTIMIX checks the types of rule predicates against the data model, and performs some type inference, inheritance information is often lacking. To support the CoSy-fSDL flatform information, the user may specify inheritance declarations in the OPTIMIX-specification himself. These declare flatform types (domains and operators) to be more specific (finer) than others. In particular this is required, when the type inference algorithm of OPTIMIX infers that two types are non-compatible which are different in the flatform but were compatible in the original CoSy-fSDL specification. E.g. consider two domains `SimpleSTMT_Assign` and `STMT_Assign`, where `SimpleSTMT` is a sub-domain of `STMT`. In the flatform the inheritance relation of `SimpleSTMT` and `STMT` is lost. If the user specifies `FINER SimpleSTMT < STMT;`, the type inference algorithm will know that both types are compatible.

Note that finer types stand to the left.

2.3.3 Data definitions with AST

OPTIMIX can collaborate with AST (or CG), the tools from the compiler toolbox Cocktail. OPTIMIX can read one or more data specification modules in AST-format and use them as definition of node and edge types. Note that either AST- or CoSy-fSDL-mode can be used, mixed mode does not work. A user may tell OPTIMIX in several ways whether/which AST-modules he wants to use.

- AST files can be handed over as normal input files, if they consist only of AST-modules. Then OPTIMIX automatically recognizes them as AST code and parses them:

  ```
  optimix optimizer.ox optimizer.ast optimizer2.cg
  ```

  Note that the first input file determines the name of the output file; thus AST-files should be given as later arguments.

- Alternatively, the user may specify one or more command line flags `-ast <file>` then this/these file(s) are parsed to find the AST data definitions. Example:

  ```
  optimix -ast optimizer.ast -ast optimizer2.cg optimizer.ox
  ```

- The user can write a `use`-clause for a file in the OPTIMIX-specification (on the same syntactic level as GRS/EARS). This clause has the form

  ```
  UseClause ::= 'use' StringList .
  ```

  `StringList` is a list of space-separated strings, which indicate file names which OPTIMIX has to open and to read. Thus, if an OPTIMIX specification is dependent on some files, the user may give these dependencies in the OPTIMIX-file. After reading the OPTIMIX-file OPTIMIX will read the files of the use clause.

In AST-mode, code is generated in plain C (without access functions). The sol-library is used for the available functors (graph and set template classes, file `sollib.ps`).

Available sol-library functors

In AST-mode OPTIMIX supports the following functors. Consider the documentation of the sol-library for more information.

```
SolFunctor ::= SolGraphFunctor | SolSetFunctor .
SolGraphFunctor ::= 'hgraph' | 'bipuni'  .
SolSetFunctor ::= 'conslist' | 'consset' | 'hashset' | 'ptrarray' | 'bitset'.
```

`conslist` provides a simple ordered list of objects (adress list). `consset` provides a linked-list based object set. `hashset` provides a set module that enters elements using a hash function on the object's address. `ptrarray` is a pointer-string set module. `bitset` is a bitset module.

`hgraph` is a simple unipartite graph with parametrizable neighbor sets. `bipuni` is a bipartite graph with parametrizable neighbor sets. All graphs of a graph functor type must currently be created 'by hand', i.e. the user must create the graph by calling a C Function which is provided by the sol-library. Nodes must be associated to the graph by calling `addnode` functions. Please consider the documentation of the sol-library for more information on how to call allocation and association functions.

The functors `hgraph` and `bipuni` are generic functors. Their modules provide create-functions which can be parametrized by the neighbor set types the graph functor should use. This may be an arbitrary set or list type from the sol-library.

2.3.4 Refining of AST field types

Unfortunately AST does not know graph-, set-, and list-fields. This means, that although existing AST-modules of compilers may be read by OPTIMIX, OPTIMIX cannot find graph and set fields in the data definition. However, set and graph fields can be introduced with two tricks:

1. The user declares the type of a scalar attribute to be of a flat sol functor application. This is an identifer consisting of the functor name and all parameters, concatenated by _, e.g.

   ```
   Object =
      [setfield: consset_Object2]
      .
   ```

   Then OPTIMIX understands that `setfield` has a type functor call to the `consset` functor and infers correctly that `setfield` may be used as predicate over `Object` and `Object2`. The disadvantage is that AST does not know anymore which type this field has; it assumes that `consset_Object2` is a scalar attribute. Thus no AST functionality on this field is available.

2. Set or graph fields may be specified in AST-modules (which only AST will read) as scalar attributes, and the OPTIMIX-user has to give an additional *refine specification* for the field in another module (which only OPTIMIX will read). If a field (or a tree module) has the AST property `REFINE` then OPTIMIX assumes that this field (resp. all fields) provide finer types for already defined fields. OPTIMIX looks up the already specified fields and changes their type to functor applications.

   Assume an AST-module which contains the AST data specification, which will only be read by AST:

   ```
   MODULE x TREE Tree RULES  /* only read by AST */
   Object =
      setfield: Object2
      .
   END x
   ```

   Then we can refine the type of `setfield` in another module, which will only be read by OPTIMIX, to a functor call using the attribute property `REFINE`:

   ```
   MODULE y1 TREE Tree RULES  /* only read by OPTIMIX */
   Object =
      (setfield: consset(Object2) REFINE)
      .
   END y1
   ```

Or we may give the property to the entire module:

```
MODULE y2 TREE Tree PROPERTY REFINE RULES  /* only read by OPTIMIX */
Object =
   (setfield: consset(Object2))
   .
END y2
```

Note that set fields have to be bracketed by `()`. Thus OPTIMIX understands a slightly extended AST language:

```
ASTObjectField ::= AstSetField .
ASTSetField ::= '(' Ident ':' FunctorCall ')'.
```

With this mechanism we can use module `x` for generation of AST code, and module `y1` or module `y2` for OPTIMIX type refinement. AST will not recognize the `REFINE` property. and will emit errors if feed by `y1` or `y2`.

Chapter 3

Specification of Graph Rewrite Systems

This section describes how graphs can be constructed and manipulated by OPTIMIX. OPTIMIX provides two kinds of graph rewrite systems for this: *edge addition rewrite systems (EARS)* and general terminating *exhaustive graph rewrite systems (XGRS, GRS)*.

EARS are equivalent to DATALOG with binary predicates [CGT89b] [CGT89a] [Aß94] [Aß95], and their rules may be written in this style (similar to Prolog clauses). In DATALOG rule tests (rule bodies) are rule right hand sides. In graph rewrite rules rule tests form left hand sides. In order to avoid confusion we will denote the left hand side of GRS rules and the right hand side of DATALOG rules with the term *rule test*. We will denote the right hand side of GRS rules and the left hand sides of DATALOG rules by *rule transformation*.

3.1 Graph rewrite specifications

A OPTIMIX-module consists of one or several graph rewrite system specifications. For each GRS one C routine is generated, having the same name. We distinguish conceptually EARS which consist of rules that only add edges (marked by the keyword `EARS`) and more general graph rewrite systems with transforming rules (marked by keyword `GRS`).[1]

The syntactic composition of EARS and GRS is the same, they consists of a parameter list, and one or several rule groups, called *strata*. The code for the stratas is generated in their specification order, one module after the other.

```
GraphRewriteSystem ::= ( 'EARS' | 'GRS') Ident '(' Parameters ')' '{' Stratum + '}'
   | ( 'EARS' | 'GRS') Ident '(' Parameters ')' Stratum  .
```

3.1.1 Strata

A stratum consists a range declaration and several graph rewrite rules.

```
Stratum ::= '{' RangeDeclarations [ Declarations ] [ Options ] [ FIRSTCode ]
   'RULES' Rules [ LASTCode ] '}'.
```

For each stratum range declarations for variables (nodes) have to be made (section 3.3). These declarations convey to which node domains the root nodes of the stratum refer [Aß94]. Also variable declarations (node declarations, section 3.4), options (section 3.6.1), FIRST- and LAST-Code may be given (section 3.6.2).

[1] This is only a syntactic distinction. Currently the keywords don't have a special meaning, but this may change in future.

An GRS or a stratum is *recursive*, if it defines a relation (an object type) which is also used (tested). Then the generated code contains a fixpoint loop to detect the fixpoint. For non-recursive strata no fixpoint loop is generated.

Each rule of a stratum leads to the generation of several rule test loops over the nodes of the mentioned graphs. How the rules are evaluated within a strata, is decided by OPTIMIX according to the evaluation strategy for GRS [Aß94].

Currently GRS(k), $k \geq 1$ are allowed. [2]

An edge-addition stratum constructs graphs by building a relation between one or two node domains, e.g. relating the node domain of a homogeneous graph or the two node domains of a bipartite graph. Each successful rule application adds one or more edges to the graph. The process can also be seen as inference of predicates between the nodes. Because an edge-addition stratum is confluent and terminating, the process stops and yields the desired graph. You may also say that edge-addition strata have a unique fixpoint. Rule syntax for edge-addition rules is given in section 3.6.

Strata that contain transforming rules need neither be terminating nor yielding unique normal forms (unique results). The syntax of transformation rules is given in section 3.7.

3.1.2 Termination check

OPTIMIX detects whether a stratum is an edge-addition stratum, and emits a corresponding information. These strata always terminate.

OPTIMIX can prove for certain other types of strata whether they terminate. This check is performed according to the *edge-additive* termination criterion of XGRS [Aß96b]. The result for each stratum is printed as information on the console. With option `ShowTermination` also the *termination edges* of each rule are printed.[3]

3.1.3 Choice-Strata

A *choice-stratum* consists of a number of rules that are tried in source order until the first redex is found. Rules are not evaluated until fixpoint; the manipulated graph is only searched until the first redex is found.

```
Stratum ::= '{|' RangeDeclarations [ Declarations ] [ Options ] [ FIRSTCode ]
   Rules [ LASTCode ] '|}'.
```

Because strata can be nested, choice-strata offer a nice opportunity to specify alternative rule conditions:

```
{  // A normal stratum with nested sub-strata
   DECLARE r:DatalogExpr;
   RULES
   // here a normal rule begins
   DatalogRuleBlock(s,r), RTG(r,rtg2),
   {| // Second level: an alternative stratum (two alternative conditions)
      // Reuse range specification, reuses outer node rtg2
      RANGE REUSE rtg2;
      // Two alternative rules:
      // The first rule tests something complex
      (  AddedNodes(rtg2,AN1), RTGNodeType(AN1,AEI3),
         GlobalClass(AEI3~Instantiated,GC3),
         GC3    ~ GlobalClass name => AN1Name,
         AN1Name != AESuccname,
```

[2] The case $k > 2$ does not occur very often, and such GRS have not been tested. It may happen that if the signatures of the rules do not overlap in list form, incorrect code is generated.

[3] The termination check is implemented with Optimix itself. Please look at the specification file `examples/termination.ox`.

```
            AN1Name != AEPredname
         ==>
         )
         (  // A very simple rule that only tests something
            rtg2 ~ RTG,
            * TEST(cglist_empty(rtg2->AddedNodes,kNoRTGNode))
         ==>
         )
      |}
   ==>
      // This is the rule action of the one rule of the outer stratum:
      // add a new node and link it
      ADD newlabel:OneIdent;.
      EdgeTerminationLabels(r,newlabel)
   .
   }
```

3.2 Parameters of routines in the generated code

For each GRS one C routine with the same name is generated. For these routines OPTIMIX generates parameter lists which consist of three subsets of parameters:

- explicitly specified parameters,
- parameters stemming from range declarations with automatic parameters,
- parameters which are graphs that are tested in rule tests or assigned in rule transformations.

If the user specifies all used/modified graphs as parameters explicitly, list (3) is empty. List (2) is only generated if declarations with automatic set parameters were specified.

Explicit parameter specification

```
Parameters ::= Parameter // ','  .
Parameter ::= Variable ':' C-Type .
```

Explicit parameter specifications serve to hand help variables over to the generated routine. They can serve to pass the engine state, or other variables that may be used in target predicates. Their type must be a C-Type (which can also be a FlatFormType).

Parameters stemming from range declarations

Each automatic range declaration (section 3.3) delivers one or two parameter declarations for the generated routine.

Parameters stemming from graph usage

Each graph tested or manipulated by a rule must be passed as parameter of the generated routine. However, the user need not provide declarations for these; OPTIMIX automatically generates a correct parameter list. The graph parameter list is ordered alphabetically.

The user has to take care that these parameter graphs are prepared correctly:

- all graphs must be created by a create-function of the functor library
- graphs must have nodes (and edges if they are not empty). The nodes must have been added to the graph by calling `addnode`-functions of the graph functors, single edges may be added by `addedge`-functions. Then more edges are calculated and inserted by the generated routine.

- if predicates are stated over the same variable, the universes of the corresponding graphs must be the same (*equality on graph node domains*). Otherwise unexpected results can occur.

3.3 Stratum range declarations

Code generation for GRS and strata relies on the concept of *GRS order* [Aß94]. The order of a stratum is roughly the same as the maximal number of root nodes in rule tests which have different types. These nodes (these variables) are called *order loop nodes*. For each order loop node OPTIMIX needs a *range decaration*, i.e. a specification to which node set the node is instantiated.

The most simple case is if the range of an order loop node is just one parameter node. Then the range declaration can be omitted, if the variable is specified in the parameter list.

```
RangeDeclarations ::= 'RANGE' RangeDeclaration + |  .
RangeDeclaration ::= ParameterRange | AutomaticParameterRange | ReuseRange .
```

If the range of the order loop node is a set or a graph node domain, a *range declaration*, i.e. a declaration to which *graph node domain* or *node set* the order loop node is initialized. Ranges can refer to user specified parameters or to automatically generated parameters.

Range declarations on user specified parameters

```
ParameterRange ::= Variable '<=' ParameterSet
   | Variable '<=' ParameterGraph ['.' 'TARGET'] .
ParameterSet ::= Variable .
ParameterGraph ::= Variable .
```

A range declaration on a user parameter consists of a specification which parameter of the GRS makes up the range of the node.

- single node parameter.
  Then no range declaration is necessary.

- node sets of graphs.
  Then the order loop range is initialized to the node domain of a parameter graph. If the modifier `.TARGET` is specified, the target node domain (domain 2) of a bipartite graph functor is taken, otherwise the source domain (domain 1).

- sets.
  If the mentioned parameter name is a set the order loop node is instantiated from this set.

Range declarations on automatically generated parameters

```
AutomaticParameterRange ::= Variable '<=' UsedGraphName ['.' 'TARGET']
   | Variable '<=' SetFunctor '(' fSDLDomain ')'
   | Variable ':' fSDLDomain '<->' Variable '<=' SetFunctor '(' fSDLDomain ')'
   | Variable ':' fSDLDomain .
UsedGraphName ::= Ident .
```

It is also possible to save the writing of parameters and let OPTIMIX infer them. Then a range declaration must give a hint of which type the node must be taken. Currently there are the following possibilities:

- specify a graph name which is used in the rules as predicate.
  Then the order loop range is initialized to the node domain of that (tested or modified) graph. Also that graph is inserted automatically in the parameter list of the generated routine.

  If the modifier `.TARGET` is specified, the target node domain (domain 2) of a bipartite graph functor is taken, otherwise the source domain (domain 1).

- sets.
  If the range is declared to be an application of a set functor, a set parameter is inserted automatically in the parameter list of the generated routine. This set is then taken to initialize the order loop node.

- single source path problem (SSPP) initialization.
  The range of an order loop node can also be only one single parameter object. Then the rule which contains the order loop node is considered to be an SSPP rule with a single source node and a result solution set which contains all nodes that fulfil the SSPP problem (section 3.8.2). The result set is thus the second part of the declaration. Source node of the SSPP as well as the result set are inserted automatically as parameters of the generated routine.

- single parameters.
  Then the order loop domain is just a variable, which is included automatically in the parameter list of the generated routine.

Range declarations on outer nodes

```
ReuseRange ::= 'REUSE' Ident
```

Within nested strata nodes from outer strata can be reused. Then it is assumed that the range of the variable is only one node, a node of an outer stratum. An example can be found in section 3.1.3.

Other examples of range declarations

```
////////////////////////////////////////////////////////////////////
//
// Range declarations
//
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
// Explicit parameter ranges
////////////////////////////////////////////////////////////////////

MODULE RangeDeclDemo OPT use "files.ast"
grs RangeDeclDemo1(g:State)
{
   RULES   // g is taken from the single node parameter
   Files(g,File), File ~ sun4 ==> Files(g,gen_lalr) .
}
grs RangeDeclDemo2(s:consset(State))
{
   RANGE g <= s;  // g is taken from the parameter set s
   RULES
   Files(g,File), File ~ sun4 ==> Files(g,gen_lalr) .
}
grs RangeDeclDemo3(h:hgraph(State))
{
   RANGE g <= h;  // g is taken from the parameter graph h
   RULES
   Files(g,File), File ~ sun4 ==> Files(g,gen_lalr) .
}
////////////////////////////////////////////////////////////////////
// Automatic parameter ranges
////////////////////////////////////////////////////////////////////
```

```
grs RangeDeclAutoDemo1()
{
   RANGE g : State; // automatic single node parameter
   RULES
   Files(g,File), File ~ sun4 ==> Files(g,gen_lalr) .
}
grs RangeDeclAutoDemo2()
{
   RANGE g <= dep2;  h <= dep2.TARGET;
      // g from source domain, h from target domain of graph dep2.
      // In this case this is the same domain, because dep2 is a hgraph.
   RULES
   sat(g,File), File ~ sun4 ==> dep2(g,gen_lalr) .
   dep2(h,g), h ~ sun4 ==> dep2(g,h) .
}
grs RangeDeclAutoDemo3()
{
   RANGE g <= consset(State); // automatic set parameter
   RULES
   Files(g,File), File ~ sun4 ==> Files(g,gen_lalr) .
}
grs RangeDeclAutoDemo4()
{
   RANGE g <= hgraph(State); // automatic graph parameter
   RULES
   Files(g,File), File ~ sun4 ==> Files(g,gen_lalr) .
}
END RangeDeclDemo
```

3.4 Stratum variable declarations

OPTIMIX infers types for variables by looking up the predicates as fields in the data model. Sometimes it cannot infer the class of a variable, it may infer different types for variables, or it may find a type which is too general.[4] Then the user can help OPTIMIX by giving additional declarations for variables. They hold for all rules of the current stratum.

```
Declarations ::= 'DECLARE' Declaration * .
Declaration ::= IdentKommaList ':' NodeType ';'.
   | ExternalFunctionDeclaration
   | ExternalFunctionDeclaration
   | ViewDeclaration .
```

A node declaration is much like a variable declaration in Modula-2. In AST-mode a class must be given as type, in CoSy-fSDL-mode domains and/or operators have to be given. OPTIMIX then incooperates these declarations into his type inference. Declarations of external functions are in section 3.6.3, view declarations in section 3.8.3.

3.5 Rules in a stratum

Rules in a stratum are specified either in the style of DATALOG (if they are edge-addition rules), or in the style of graph rewrite rules. Each rule consists of a rule test part and a rule transformation part. Edge-addition rules only allow predicates in their transformation part. Each rule may be accompanied by rule options, FIRST- and LAST-code. Rules are either enclosed in '(' ')' brackets or they end with a '.'. Facts are explained in section 3.8.1, transformation rules in section 3.7.

```
Rules ::= (EARSRule | XGRSRule | EARSFact) +  .
EARSRule ::= [ Options ] [ FIRSTCode ] EdgeAddition ':-' RuleTest [ LASTCode ] '.'
   | '(' [ Options ] [ FIRSTCode ] EdgeAddition ':-' RuleTest [ LASTCode ] ')'.
```

[4] This may happen especially in CoSy-fSDL-mode because here the code generation needs not only domains, but also operators to generate access functions.

```
RuleTest ::= Predicates .
EdgeAddition ::= Predicates .
Predicates ::= Predicate // ',' .
```

3.6 Rule tests

3.6.1 Options for strata and rules

```
Options ::= 'OPTIONS' Name + '.'.
Options ::= '[' Name // ',' ']'.
```

Strata and rules may also be annotated with an option list (*options*). This is a list of strings, enclosed in square brackets `[]` or appended after the keyword `OPTIONS`. If such an option is set, the semantical analysis, optimization and code generation phases of OPTIMIX are performed in a stratum/rule specific way.

Current available stratum options are:

- **CentralNeighborSetComparison, LocalNeighborSetComparison, DirectFixpointCheck**
  OPTIMIX knows three kinds of fixpoint detections for edge additive rules: central neighbor set comparison, local neighbor set comparison, and direct fixpoint check (section 3.9). Normally central neighbor set comparison is prefered over local neighbor set comparison. The flag can be used to override this default.
- **NoFixpoint, Fixpoint** Do not generate a fixpoint loop/generate a fixpoint loop for the stratum.

Current available rule options are:

- **JOIN** Use join code generation mode, even if on-the-fly was analysed.
- **ETJoin** (alpha-tested) Use element-test join code generation mode. This mode uses element tests instead of join equality conditions. It should for a lot of cases speedup the join, if element tests of the last functor of a path are possible in constant time.
- **ETFilters** (alpha-tested) Use element-test path pre-filtering. Paths are generated several times, the first times for pre-selections.
- **LocalTests** Perform the pattern matching on a node always, if an instance of the node is traversed. This option results in more pattern matching tests, but fewer traversals, because the join search space of path problems is diminished.

3.6.2 FIRST and LAST target predicates for strata and rules

Strata as well as rules can be annotated by a FIRST and an LAST target predicates. The code of FIRST predicates is printed in the generated code right after the variable declarations for a stratum (or a rule). Here the user can define his own variables for use in target predicates. The code of the LAST target predicate before the stratum/rule end.

```
FIRSTCode ::= 'FIRST' TargetPredicate .
LASTCode ::= 'LAST' TargetPredicate .

 FIRST {* /* here is FIRST */ *}
    LinearBlocks(PBody,B), Stmts(B,Ass), Ass ~ mirAssign
 ==>
    AllDefinitions(PBody,Ass)
 LAST {* /* here is LAST */ *}
 .
```

3.6.3 Predicates in rules

A rule test part contains a number of predicates, which are tested against the manipulated graph. They may have the following forms:

```
Predicate ::= PredicateName '(' Pattern ',' Pattern ')'
   | 'FORALL' Variable ':' Predicate
   | 'NOT' Predicate
   | '?' ProcedureCall
   | TargetPredicate
   | TargetCodeLine
   | PatternMatchStatement
   | EqualityTest
   | FailStatement
   | Cut
   | Strata .
PredicateName ::= Ident [ '@' NodeType] [ '.' GraphFieldModifier] [ '.' OrderIndicator ] .
```

Simple predicates

Simple predicates are always binary because they refer to graphs. Simple predicates contain patterns or variables as arguments. Predicate names must exist as the name of a object field in the data model (in CoSy-fSDL in an operator in a certain domain). Thus a predicate in a rule test or rule transformation refers to

- a field which has the type of a graph functor call (graph field)
- a field which has the type of a set/list functor call (set field)
- a field which has the type of a simple domain (non-graph) (pointer field).

The predicate

```
.. :-   p(X,Y),..
```

is true if the object `Y` is contained in the set `X.p`. Also (in the generated code) the predicate `p(X,Y)` delivers all objects `Y` which are linked under field (or graph) `p` to object `X`.

Type inference for simple predicates

OPTIMIX looks up the field name in the data model and annotates with the predicate a set of types (in CoSy-fSDL operator/domain pairs). This is a set of types because a field can turn up in several objects. In CoSy-fSDL also operators may be contained in several domains. These sets of alternative types are then intersected and unified against each other during the type inference. OPTIMIX always tries to retain *finer* types, i.e. more specific types, which then provides better information for code generation. The rules according two types are compared are the following:

- an AST type is finer than its superclasses.
- an CoSy-fSDL operator is finer than a containing domain.
- an CoSy-fSDL operator/domain pair is finer than the domain.
- a type is finer than another if it has been declared so in a FINER inheritance declaration.

At the end of the type inference process there should be unique types for all variables in rules. If not, OPTIMIX will prompt an error. Either this is a real typing error or the user can give more type information to OPTIMIX by providing inheritance declarations (section 2.3.2) or variable declarations (section 3.4).

In CoSy-fSDL-mode however, this scheme currently has one restriction. If a field is contained in several operators, and is not a shared field, then the user has to specify with the field a domain/operator specification. E.g. in the CCMIR the field `Then` occurs in operator `mirIf` as well as in `mirTryAcquire`. A predicate using it in domain `mirIf` should look like:

```
Then@mirSimpleSTMT@mirIf(Stmt, ThenPart)
```

If the field is a shared field between all operators that use it, the field alone is sufficient as predicate name.

The ANY class Users can specify nodes to be of class `ANY`. This type is coarser than all other types, and all types are compatible with it. If AST-mode is used, also the type `tTree` can be used, which is equivalent.

Graph field modifiers

```
GraphFieldModifier ::= 'succ' | 'pred'.
```

Graph field modifiers serve to indicate which kind of functions should be called to navigate in the generated code. From OPTIMIX's point of view a functor-created graph defines two default fields for the parameter domains of the functor application. These two default fields can be used as predicates in clauses. For instance, if we have the following CoSy-fSDL definition

```
domain Proc : {
   Procedure < BlockGraph: SGRAPH(BasicBlock) >
> }
domain BasicBlock : {
   BasicBlock <  >
> }
```

then the functor call `BlockGraph: SGRAPH(BasicBlock)` creates for domain `BasicBlock` two default fields `BlockGraph` and `BlockGraph.pred`. These field names denote all successors resp. predecessors of a `BasicBlock` concerning the functor-created graph `BlockGraph`. With `p.succ` or `p` the successor relation of graph `p` is denoted, with `p.pred` the predecessor relation is denoted.

Order indicators

```
OrderIndicator ::= 'first' | 'last' | 'next' | 'prev' | 'before' | 'after' | 'any'.
```

If users specifies predicates that refer to fields of list-functor type, special *order indicators* can be used to find out certain special elements of the list. The order indicators generate an access to specific list elements. There are the following types, exemplified by a relation `Stmts`, the statements of a block:

- `Stmts.after(Block,S1,S2)` generates a loop over all successors `S2` of `S1` in the `Stmts` of `Block`.
- `Stmts.before(Block,S1,S2)` generates a loop over all predecessors `S2` of `S1` in the `Stmts` of `Block`.
- `Stmts.next(Block,S1,S2)` generates an access to the successor `S2` of `S1` in the `Stmts` of `Block`.

- `Stmts.prev(Block,S1,S2)` generates an access to the predecessor `S2` of `S1` in the `Stmts` of `Block`.
- `Stmts.first(Block,S1)` generates an access to the first element `S1` in the `Stmts` of `Block`.
- `Stmts.last(Block,S1)` generates an access to the last element `S1` in the `Stmts` of `Block`.
- `Stmts.any(Block,S1)` generates an access to an arbitrary element `S1` in the `Stmts` of `Block`. For lists the head of the list is taken. This indicator can also be applied to set functors. Then it emits a call to the choose-function of the set functor.

The variable `Block` must be defined earlier in the specification. Note that the arity of several ordered predicates is 3.

```
////////////////////////////////////////////////////////////////
//
// Test list functor addition predicates in rule transformations
//
////////////////////////////////////////////////////////////////
MODULE AdditionDemo OPT use "files.ast"
grs AdditionDemo()
{
   RANGE g: State; RULES
      Files(g,File), sat(File,Fx), sat(Fx,Fy), dep(File,OFile)
   ==>
      dep(File,OFile),
      dep.first(File,OFile),
      dep.last(File,OFile),
      dep.after(File,Fx,OFile),
      dep.before(File,Fx,OFile),
      dep.next(File,Fx,OFile),
      dep.prev(File,Fx,OFile),
      dep.any(File,OFile),

                /* test cglist functor */
      dep3(File,OFile),
      dep3.first(File,OFile),
      dep3.last(File,OFile),
      dep3.after(File,Fx,OFile),
      dep3.before(File,Fx,OFile),
      dep3.next(File,Fx,OFile),
      dep3.prev(File,Fx,OFile),
      dep3.any(File,OFile),

                /* test cgthrlist functor */
      dep4(File,OFile),
      dep4.first(File,OFile),
      dep4.last(File,OFile),
      dep4.after(File,Fx,OFile),
      dep4.before(File,Fx,OFile),
      dep4.next(File,Fx,OFile),
      dep4.prev(File,Fx,OFile),
      dep4.any(File,OFile)

                .
}
END AdditionDemo
```

All-quantified predicates

Normally all predicates are existentially quantified in their variables. However, one predicate in a rule is allowed to be preceeded by an all quantifier, e.g. `FORALL V: p(X, V)`. The all-quantified variable must be the right variable of the predicate. Predicates in the head of a rule cannot be all-quantified. Currently the concept of all-quantifiers is rather restricted. It works only in two situations:

- The all-quantified variable is the middle variable of a path with two predicates and the rule test is a single path. This is the standard situation for MUST dataflow analyses (section 5.2.2).

- The all-quantified variable is the sink of an rule test graph. This rule test graph must also be either a path or a dag (see example specification `copyprop.ox`).

Negated predicates

If a predicate is preceeded by a `NOT`, it is negated. Then the generated code tests that no edge between the two variables of the predicate exists, and skips all combinations of nodes which are linked by a corresponding edge. Negation is allowed in the following contexts:

- In rule tests if predicates are used that are graph functor instantiations. Negation can only be performed if a universe is known against the completion of a set of nodes is performed. This is the case only for graph functors, in which the set of graph nodes represents this universe.

  Negation is performed by a loop over the universe, skipping those nodes which are the neighbor set of the predicate.

- In rule tests for bitset predicates. They also have a universe which consists of all nodes the bits refer to. Negation is performed by a bitset complement.

- In rule transformations. Then in the code an edge of the denoted graph is deleted, not added.

Checked calls to external predicate functions

```
ProcedureCall ::= Ident '(' ActualParameter // ',' [ '==>' ActualParameter // ',' ] ')'.
```

If a predicate starts with a `?`, then OPTIMIX assumes that the rest is a call to a C function returning a boolean. Thus it generates this call and checks its result with TRUE (integer constant 1). If the called predicate fails, also the rule fails. Otherwise the rule test is continued.

The list of actual parameters to a call must consist of simple variables. There is an IN parameter list (before the `==>`) and an OUT parameter list (after the `==>`). The IN parameters are considered to be pattern variables which are handed over to the called routine. The OUT parameters are also handed over as reference parameters, i.e. their addresses are handed over.

Calls to external functions

Edges in graphs may be simulated by calling external C functions. The name of the functions must appear in a DECLARE declaration:

```
ExternalFunctionDeclaration ::= IdentKommaList ':' 'fun' NodeType '->' SetOrNodeType
ExternalProcedureDeclaration ::= IdentKommaList ':' 'proc' NodeType '->' SetOrNodeType
```

The first alternative declares an external C function returning a node type or a set over a node type. The second alternative declares a void C procedure with two parameters, the first an input parameter (pointer/object type), the second one an output set parameter (set type). Examples of declarations:

```
DECLARE
   f1:      fun file -> file;
   f2, f3:  fun file -> conslist(file);
   f4:      fun file -> consset(file);
   p1:      proc file -> file;
   p2,p3:   proc file -> consset(file);
```

`f1` – `p3` may all be refered to as binary predicate names in rule tests:

```
( f1(01,02), p3(02,03) ==> a_normal_graph_edge(01, 03) )
```

The semantics of external C function calls in rule transformations is not defined.

```
////////////////////////////////////////////////////////////
//
// Test external function calls
//
////////////////////////////////////////////////////////////
MODULE FunctionDemo OPT use "files.ast"

grs ExternalFunctionDemo()
{
   RANGE g: State;
   DECLARE
      procedge1:    fun file -> consset(file);
      procedge2:    proc file -> consset(file);
      procedge3:    fun file -> file;
      procedge4:    proc file -> file;
   RULES
   Files(g,File),
   procedge1(File,Fx),
   procedge2(Fx,Fy),
   ? call_external(Fy ==> Fz,Fy),
   ? call_external(Fy, Fz ==> Fz),
   procedge3(Fy,Fz),
   procedge4(Fz,Fa),
   dep(Fa,OFile)
==>
   sat(File,OFile) .
}

END ExternalFunctionDemo
```

Target code predicates

```
TargetPredicate ::= '{*' any '*}'.
```

It is possible to specify C target code as predicate (a *target predicate*). This code is copied unchanged to the generated file. Target predicates always succeed.

Target predicates normally are attached to their preceeding predicates. They are copied after the code that was generated for their preceeding predicate, i.e. normally in a loop which was caused by that predicate. There are some special cases:

1. If a target predicate appears as first predicate of a rule test part, it is copied to the generated file as last action of the rule test part, but *before join conditions are evaluated.*[5]

2. If a target code predicate appears in the rule transformation of a rule it is printed after the addition/deletion of the preceeding edge in the innermost rule test loop.

3. If it appears as first predicate in a rule transformation, it is printed before the addition of the edges and the rule transformation.

4. If it appears as last predicate in a rule transformation, it is printed before the addition of the edges and the rule transformation.

5. FIRST and LAST target predicates are copied to places before and after the execution of a rule (section 3.6.2)

Target code lines

If a target predicate consists of one line of C, there is a special syntactic alternative for it. Target code lines consist of arbitrary C text, terminated by a newline character. If necessary, the newline also simulates a ','-token to the parser, so no intermediate commas are necessary with target code lines.

[5] This semantics is weird and probably will change.

```
TargetCodeLine ::= '*' any Newline .
```

Target predicates may be used for test and set of node attributes or debugging generated code. E.g. the following rule tests whether a node is marked as deleted and removes it from some graphs:

```
GRS DeleteFromStatementLists(Proc:ProcGlobal)
{
   RULES
      Body(Proc,PBody),
      {* /* This target code is printed after the code for  */
         /* the preceeding predicate Body                   */ *},
      LinearBlocks(PBody,Block),
      * /* and this here is a single line of target code */
      Stmts(Block,Ass),
      Ass ~ Assign{},
      {* /* target predicate to test, whether a node was really deleted */
         /* is copied to the rule test after the pattern match on Assign */
         if (!SimpleSTMT_Assign_get_deleted(Ass))
            continue;
      *}
   ==>
      DELETE Ass FREE;.       // really deallocate Ass
      * printf("deleting copy statement %s",STMT_provide_label(Ass));
      NOT Stmts(Block,Ass),
      NOT list_of_definitions(Proc,Ass)
      * /* This here is really the end of the rule */
   .
}
```

Pattern match predicates

```
PatternMatchStatement ::= Variable ('~' | '!~') Pattern .
```

As predicates also pattern match statements on rule test nodes (rule test variables) are allowed. If a variable is linked to a pattern with `~` this pattern match statement succeeds if the variable has the form of the pattern. If a variable is linked to a pattern with `!~` the pattern match statement succeeds if the variable has not the form of the pattern.

```
N ~ Block          // node N matches type Block
N !~ Block         // node N is not of type Block
```

In rule transformations pattern matching is not allowed.

Patterns In pattern match statements or in predicates of rule tests patterns may appear.

```
Pattern ::= NodeType
   | NodeType '{' InnerPattern // ',' '}'.
InnerPattern ::= FieldName '=>' Pattern
   | FieldName '=>' Variable ('~' | '!~')  Pattern
   | FieldName '=>' Variable  .
```

Variables in patterns are arbitrary identifiers, contary to Prolog, where each variable has to begin with a capital letter. Note that OPTIMIX only performs pattern matching, not unification. There are two kinds of patterns: outer patterns are allowed in pattern match statements, where they match already defined variables. They are also allowed in left or right parameters of simple predicates, however, only at the outer level.

Inner patterns are allowed to occur only in an outer pattern or another inner pattern. They perform field pattern matching and also variable assignment. Positional pattern matching is not possible, only

matching with a field name is allowed. Variable assignment assigns a variable to the field, if the pattern match was successful. If no variable assignment is given, OPTIMIX assigns a temporary variable to the successfully matched subtree.

For instance, the pattern match

```
S ~ If{Then => A ~ Assign}
```

tests whether a variable S consists of a `If` where the field `Then` is a `Assign`. The variable `A` is assigned to the assignment statement.

Restriction of the current implementation: Note that the variables which are defined in patterns are not allowed to be used for further navigation, only for the use in target predicates, e.g. to test attributes. Also it is not allowed to use constant patterns. Constant comparisons are only allowed in equality tests (section 3.6.3).

Equality tests

```
EqualityTest ::= PatternVarEqualityTest | RTGNodeEqualityTest .
PatternVarEqualityTest ::= Variable BoolOp Variable .
RTGNodeEqualityTest ::= Variable EqualOp Variable .
BoolOp ::= EqualOp | '<' | '>' | '<=' | '>='  .
EqualOp ::= '==' | '!='  .
```

On pattern variables or on rule test graph nodes equality tests may be performed. For pattern variables they lead to the generation of equality/inequality functions of the opaque types of the attributes. For rule test graph nodes, in AST-mode pointer equality is used. In CoSy-fSDL-mode `DMCP_equal` is used.

```
SpecialBlocks(b,b1) :-
   BlockGraph(b,b1),
   b != b1,            // test pointer inequality of b and b1
   b  ~ Block  number => Number ,
   b1 ~ Block  number => Number2 ,
   Number <= Number2   // compare attributes Number and Number2
   .
```

3.6.4 Nested strata

In lieu of predicates strata may appear (*nested strata*). This can be used to test alternative conditions for rules (with choice-strata), or to do intermediate actions during the test of a rule.

3.7 Transformation rules

Transformation rules use the same syntax as edge-addition rules to specify preconditions, and have an additional transformational part. This transformational part consists of node deletions, node additions, edge deletions and edge additions, also to the newly created nodes.

```
XGRSRule ::= [ Options ] [ FIRSTCode ] RuleTest '==>'
   [ [ NodesToBeDeleted ] [ NodesToBeAdded ] '.' ] Predicates [ LASTCode ] '.'
   | '(' [ Options ] [ FIRSTCode ] RuleTest '==>'
   [ [ NodesToBeDeleted ] [ NodesToBeAdded ] '.' ] Predicates [ LASTCode ] ')'.
```

Strata, strata options, rule options, FIRST- and LAST-Code behave in the same way as with EARS rules.

Note that currently the user himself has to guarantee the termination of a XGRS. There is no automatic check for that, neither a test for confluence. See also the article on XGRS [Aß96a], whose method is currently not implemented..

Node addition

```
NodesToBeAdded ::= 'ADD' VariableDeclarations  .
```

Nodes which are added by the rule, have to be declared in a similar way as rule local variable declarations. In CoSy-fSDL-mode it is necessary to specify both a domain and operator for new nodes. Otherwise the correct node allocation function call cannot be generated.

Node deletion

```
NodesToBeDeleted ::= 'DELETE' IdentKommaList DeleteProperty * ';'.
DeleteProperty ::= 'MARK' | 'FREE' | 'REMOVE' | 'DELAYEDREMOVE'.
```

Nodes from the rule test which have to be deleted are specified after the keyword `DELETE`. The deletion can be done in four modes, which can be combined, e.g. it is possible to specify `MARK REMOVE` with some nodes.

The *mark mode* just markes the nodes, which are in a successfully matched redex, by setting the field `deleted`. This is a field which the user has to add to all types of objects which have to be deleted. Once the nodes are marked like this, they can be recognized as being invalid. Marking is necessary when a node belongs to a lot of graphs, not only those that were tested in the rule. Then subsequent passes over these graphs can remove all incident edges, and in the last pass also the node can be deallocated.

The *remove mode* does not deallocate the nodes but removes the node from all the containing graphs which are mentioned in the rule test. Thus it deletes all incident edges of graphs of the rule test. There still might be other graphs the node belongs to.

The *delayed-remove mode* is special. It generates a second, artificial GRS. This GRS walks the graphs of the rule test a second time, tests on deleted (marked) nodes and then performs removal of incident edges.[6] In CoSy-fSDL-mode, the walking is done via ITERLIST-LOOPs, not with LIST-LOOPs. This is due to a restriction of the fSDL-LIST functor which could not delete nodes from lists when walking the lists themselvs via LIST-LOOP.[7]

The *free mode* really deallocates the nodes. In AST-mode a function `Tree_delete` is called.[8] An appropriate macro or function has to be provided by the user. In CoSy-fSDL-mode `<domain>_delete`. is called. This function is always part of the DMCP interface.

Addition of edges to new nodes

The rule transformation part following the declarations consists again of a sequence of predicates. Here they specify edge additions and deletions. Edge additions are performed by non-negated simple predicates and may refer to new nodes as well as to old nodes. Edge deletions are performed by negated simple predicates and can of course only refer to items from the rule test.

```
(  LinearBlocks(PBody,Block),
   Stmts(Block,Assign),
==>
   ADD Assign2;.            // allocate Assign2
   Stmts(Block,Assign2),    // enter Assign2 in statements
   NOT Stmts(Proc,Assign)   // remove Assign from statements
)
```

[6] The generated GRS should only containing the rule in question. Currently it contains all rules.
[7] It may be that in the current version this is obsolete; so also delayed-remove mode is obsolete.
[8] Currently the AST tree is assumed to have name Tree. This should change

3.8 Other kinds of rules

3.8.1 Non-ground facts

```
EARSFact ::= [ Options ] [ FIRSTCode ] Predicates [ LASTCode ] '.'   .
```

In OXDML non-ground facts may be specified analogously to Coral [RSS92]. Facts are edge-addition rules without preconditions. Non-ground facts are facts that contain variables. Non-ground facts serve to initialize a graph with certain values before other rules manipulate the graph. This can be used especially for data flow analysis: the initialization statements there are non-ground facts. Non-ground facts in a stratum are always evaluated before other rules of the it are evaluated.

As example consider the specification of available expression dataflow analysis, the first two rules are non-ground facts:

```
// Find available expressions
EARS AvailableExpressions ()
{
   RANGE b <= AVIN; e <= AVIN.TARGET;

   // non-ground facts: initiallization to FULL set.
   AVIN(b,e).
   AVOUT(b,e).
   // EARS rules.
   AVIN(b,e)  :- FORALL p: BlockGraph(b,p), AVOUT(p,e).
   AVOUT(b,e) :- COMPOUT(b,e).
   AVOUT(b,e) :- TRANSP(b,e), AVIN(b,e).
}
```

Also *self-edge facts* may be specified which draw self edges on nodes:

```
EARS ComputeDominators(b: BasicBlock)
{
   Dominators(b,b).    // self-edge fact: each block is dominated by itself
   ..
}
```

Non-ground facts also may be negated. Then OPTIMIX generates loops over the graph nodes that delete existing edges.

There are all in all several possibilities, how to initialize a graph:

- make a full graph with a non-ground fact.
- make a graph with self edges with a self-edge fact.
- delete all edges in a graph by a negated fact.
- delete all self edges by a negated self-edge fact.

Before and after facts target predicates can be written. If a target predicate is written before the fact, it is copied directly before the edge addition. If it is written after the fact, it is copied direcly after the edge addition.

For further examples on facts consider example file `facts.ox`.

3.8.2 Single source path problems (SSPPs)

There is a special variant of EARS which can solve single source path problems (SSPPs) [Tar81]. An SSPP is a path problem in a graph which is described by a path expression (or a set of predicates, like in EARS) and which is applied to *one single* source node of the graph. It delivers all nodes which are

reachable from the source node under the predicates (the path expression). These nodes are called *result set*.

A GRS may contain several SSPP rules. The the source node of the SSPP and the result set of such a rule can be declared with a range declaration (section 3.3). The node is then initialized to the corresponding parameter of the generated routine, and the parameter set of the range declaration is used as the result set. SSPP rule tests are not generated among those rule tests which result from normal rules (in the order loops). Instead they are extracted and printed after them.

The following example solves a SSPP for a procedure and all its statements. It collects all assignments that are in the blocks' statement lists.

```
EARS PrepareReachingDefinitions()
{
   RANGE Proc <= ProcGlobal <-> list_of_definitions: SET(STMT);
   RULES
   list_of_definitions(Proc,Ass) :-
      Body(Proc,PBody),
      LinearBlocks(PBody,Block),
      Stmts(Block,Ass),
      Ass ~ Assign{}
   .
}
```

SSPP rules can also be used nicely to write down walking e.g. over statement lists and perform actions on them. The following example collects all assignment statements in a parameter set `list_of_definitions`. They also are entered into a global class of definitions for objects, with a target predicate side effect action. This global class is attached to global state handle (e.g. of a CoSy engine), and must be passed as parameter to the generated routine `CollectAssigns`.

```
EARS CollectAssigns(state: StateType)
{
   RANGE Proc <= ProcGlobal <-> list_of_definitions: SET(STMT);
   RULES
   list_of_definitions(Proc,Ass),
      {* EnterInDefinitionClasses(state,Ass); *}
   :-
      Body(Proc,PBody),
      LinearBlocks(PBody,Block),
      Stmts(Block,Ass),
      Ass ~ Assign{}
   .
}
```

3.8.3 View rules

(Only alpha tested)

View rules are single source path problem rules which consist of a linear chain of predicates and one single assigned predicate. In order to faciliate the writing of rule tests, these assigned predicates can be used as abbreviation for the complete view rule. Thus, if the user uses in a rule test an assigned predicate of a view rule, the rule test part of the rule is *expanded* in-line into the using rule. This works without problems because the view rule is only allowed to have chain form.

In order to define view rules the user must specify the assigned predicate in a `DECLARE` specification as follows:

```
ViewDeclaration ::= IdentKommaList ':' 'view' BinaryPredicateType ';'
BinaryPredicateType ::= NodeType '->' SetOrNodeType .
SetOrNodeType ::= FunctorCall | NodeType .
```

Rules which define this assigned predicate are assumed to be view rules and are automatically inline-expanded in other rules. Note that only one view rule per view assigned predicate may exist.

Global sets which assemble the assigned predicate elements can be choosen from any set or list functor.

```
DECLARE
   // view rule declarations
   viewedge1: view Block -> conslist(Block);
   viewedge2: view Block -> conslist(Block);

RULES
// view rule definitions.  They do not lead to code!
viewedge1(B1,B2) :- base(B1,B3),base(B3,B2).
viewedge2(B1,B2) :- base(B1,B3),base(B3,B4),viewedge1(B4,B2).

// rule with view edge call. View edges are expanded.
LinearBlocks(Proc,Block), viewedge2(Block,Block2), dep(Block2,Block3)
==>
base(Block,Block3)
.
```

3.9 Fixpoint checks

OPTIMIX knows three kinds of fixpoint detection for strata: *direct fixpoint check*, *central neighbor set comparison*, and *local neighbor set comparison*. These detection methods have different runtime costs. Not all of them are apt for all XGRS.

The first method is probably the fastest. It can be used if the functor functions which are used to perform graph manipulations, give back a change flag, if something has changed. Then functor functions for edge addition are queried if they have changed something. Direct fixpoint checking is also chosen, if no rule of the stratum is edge additive, i.e. if all rules manipulate nodes.

Central neighbor set comparison runs a bit slower. It compares the neighbor sets of order loop nodes before and after a fixpoint loop. It memorizes the old values of neighbor sets of order loop nodes by copying or assignment.

If an assigned edge of a stratum does not start at an order loop node, local neighbor set comparison is performed. This is probably the slowest mode: It copies/memorizes old values of neighbor sets each time the source node of the assigned edge is traversed.

Option flags can be used to override the default modes (section 3.6.1).

Chapter 4

Meta-Optimizations for XGRS code generation

OPTIMIX knows how to optimize the evaluation of several kinds of specifications. In order to avoid confusion with program optimization we call this *meta-optimization*. However, note that all this may be topic of implementation restrictions, see section 6.1.

4.1 Bitset optimization

A rule is bitset optimizable, if all incoming edges to the target node of an assigned edge are implemented by bitsets. If so, for the rule bitset operations are generated; otherwise normal object-based code generation is used.

The following rule is bitset optimizable, if we suppose that `EC_DSAVE_OUT`, `EC_DSAVE_IN`, and `EC_TRANSP` are implemented as `SETF(Stmt)`. The rule test graphs is a directed acyclic graphs, all assigned edges are added in forward direction, and all edges incoming to the target node of the assigned edge `s` are have bitset functors..

```
EC_DSAFE_IN(Block,Expr) :- EC_TRANSP(Block,Expr), EC_DSAFE_OUT(Block,Expr).
```

The generated code is

```
for (_onceindex = 1; _onceindex > 0; _onceindex--)
{
   /* path test [Block]-EC_TRANSP->[Expr] */
   Block = OLoopRawNode_0;
   /* bitset predicate test EC_TRANSP  32, 44 */
   /* if result set not already created, create it */
   if (Result_Block_EC_TRANSP_Expr == NULL)
      Result_Block_EC_TRANSP_Expr = SETF_mirEXPR_EC_create(mirBasicBlock_mirBasicBlock_get_EC_TRANSP(Block),
            SETF_init_full);
   else
      SETF_mirEXPR_EC_full(Result_Block_EC_TRANSP_Expr);
   SETF_mirEXPR_EC_inter2(Result_Block_EC_TRANSP_Expr,mirBasicBlock_mirBasicBlock_get_EC_TRANSP(Block));
   SETF_mirEXPR_EC_inter2(Result_Block_EC_TRANSP_Expr,mirBasicBlock_mirBasicBlock_get_EC_DSAFE_OUT(Block));
   SETF_mirEXPR_EC_union2(mirBasicBlock_mirBasicBlock_get_EC_DSAFE_IN(Block),Result_Block_EC_TRANSP_Expr);
   /* end path test [Block]-EC_TRANSP->[Expr] */
} /* end of rule fake for-loop */
```

4.2 Bidirectional edge optimization

If a functor provides bidirectional implementation (e.g. `hgraph`), OPTIMIX can use both directions for code generation. During the computation of the edge-disjoing path cover of the rule test graphs, OPTIMIX selects one of the directions. Thus bidirectional edge optimization completes bidirectional graph functor edges in order to find better paths for code generation.

Thus the order of a rule test graph can be reduced to 1, if enough bidirectional functors are used.

4.3 Index edge optimization

(only tested in CoSy-fSDL-mode)

If an XGRS has order 2, and uses attribute equality tests on its rule test root nodes, it can be speed up by the use of index structures.

```
IndexDeclaration ::= 'INDEX' Variable ':' IndexName [ 'FUNCTION' IdentList ] ';'.
IndexName ::= 'HASHTABLE' | 'PLAINTABLE'.
```

If an index is specified on a variable, the code generation changes as follows. First the order loop node domain of the specified variable is traversed to collect all objects into the index structure. Then the index is used as *virtual edge* between the two root nodes of the rule. This virtual edge is traversed during rule test. The rule (and may be the stratum) gets order 1 and will be generated with other rules that have the same (single) range.

Currently there are hash tables (multi-valued index) and plain pointer tables (one-valued).[1] OPTIMIX emits calls to C modules which both can be found in the sol-library.

```
EARS EquivalenceOfExpressions()
{
   RANGE i1 <= consset(Expression); i2 <= consset(Expression);
   INDEX i2: HASHTABLE  FUNCTIONS hash_Expression, compare_Expression;
   RULES
   simple_equal(i1, i2) :-
      Type(i1,T1),Type(i2,T2),  // two rule test root nodes: order 2
      T1 == T2,
      i1 ~ IntConst Value => V  ,
      i2 ~ IntConst Value => V1 ,
      V == V1              // attribute equality test
      .
}
```

transforms logically into

```
EARS EquivalenceOfExpressions()
{
   RANGE i1 <= consset(Expression); RULES
   simple_equal(i1, i2) :-
      VirtualIndexEdge(i1,i2),   // one rule test root node: order 1
      Type(i1,T1),Type(i2,T2),
      T1 == T2,
      i1 ~ IntConst Value => V  ,
      i2 ~ IntConst Value => V1 ,
      V == V1
      .
}
```

The index function may be accompanied with the following functions in order:

1. hash function: Hashes an object into a `int` hash value. C signature:

   ```
   int hash(<NodeType> n);
   ```

2. compare function: compares two objects on equality. Should behave like `strcmp`: give back 0 if object 1 is equal to object 2, -1 if smaller, 1 if greater. Example:

   ```
   int compare_Expression(Expression e, Expression e2)
   {
      if (e->Kind == e2->Kind)   return 0;
   ```

[1] Only hash tables are tested yet.

```
    if (e->Kind >  e2->Kind)   return 1;
    return -1;
}
```

If one of them is left out, OPTIMIX chooses a standard function (probably behaving inefficient). Choosing appropriate hash functions is quite important. Also note that the hash function has to be specified anyway: if it is left out, a dummy void function has to be specified instead.

Chapter 5

Examples and Miscellaneous

5.1 AST/standalone-mode examples

Please refer to the example files in the subdirectories `$OPTIMIXDIR/demo/examples` and `$OPTIMIXDIR/demo/ast`. The first directory contains several examples of this manual. The second directory contains the following examples:

ccmir.ast A fragment of a real-life intermediate language: the CCMIR (Common COMPARE Medium Intermediate Representation)

files.ast A fragment of some specification for file of different types.

reach.ox Reaching definitions for CCMIR.

example-reachable.ox Transtive closure.

exprtab.ox Expression equivalence for CCMIR.

dominators.ox Computing dominators on block graph.

facts.ox Example facts.

livecopies.ox Live copy information.

copyprop.ox Copy propagation transformation.

5.2 CoSy-fSDL-mode examples

Here we will present some short examples for CoSy-fSDL mode. We assume a basic block graph is defined in a procedure as follows. In CoSy this can be done in a view specification of an engine. We that the basic block graph has already been constructed and entered into relation `BlockGraph`.

```
domain mirProcBody <
      BlockGraph:      SGRAPH(mirBasicBlock),
      ReverseBlocks:   SGRAPH(mirBasicBlock),
      ReachableBlocks: SGRAPH(mirBasicBlock),
      Dominators:      SGRAPH(mirBasicBlock),
      SelfDom:         SGRAPH(mirBasicBlock),
      USED:            BIPUNI(mirBasicBlock,mirLocal),
      Livein:          BITUNI(mirBasicBlock,mirLocal,Livein),
      Liveout:         BITUNI(mirBasicBlock,mirLocal,Liveout)
   >;
```

Then we may write the following specifications.

```
/* Compute the inverse of the basic block */
EARS ComputeReverse()
{
   RANGE b <= BlockGraph;        // implicit parameter graph BlockGraph
   RULES
      ReverseBlocks(b,b1) :- BlockGraph(b1,b).
}
```

This EARS of order 1 just builds up the reverse basic block graph, a relation `ReverseBlocks`. The automatic-parameter-range declaration tells that the order loop variable is to be initialized from the node domain of graph `BlockGraph`. The following shows how the generated routine may be called from C code (CoSy):

```
BlockGraph = mirProcBody_get_BlockGraph(procbody);
// or:
// BlockGraph = SGRAPH_mirBasicBlock_create();
//     add also some nodes with addnode functions..
mirProcBody_set_ReverseBlocks(SGRAPH_mirBasicBlock_create());
ReverseBlocks = mirProcBody_get_ReverseBlocks(procbody);
CopyNodes (BlockGraph, ReverseBlocks);  // should copy the nodes of the SGRAPH
ComputeReverse (BlockGraph,  ReverseBlocks);
```

The order of the parameter graphs to `ComputeReverse` is alphabetically.

The next example computes dominator analysis. The first stratum initializes. Initially all nodes dominate all others except that the entry node does not dominate anyone.

```
EARS ComputeDominators()
{
   RANGE b <= Dominators; b1 <= Dominators;
   RULES

   Dominators(b,b1) :- BlockGraph.pred(b,PredecessorBlock).
                       // initially a node dominates each other node.
                       // The dominators of the entry node, however, are left empty.
   SelfDom(b,b).       // this predicate is used for adding each node to a
                       // set Dominators during the processing in ComputeDominators
}
{
   RANGE b <= BlockGraph;
   RULES

   // a node dominates another if all predecessors dominate the other
   Dominators(b,b1) :- FORALL p: BlockGraph.pred(b,p), Dominators(p,b1).
   Dominators(b,b1) :- SelfDom(b,b1).
}
```

`BlockGraph.pred(b,p)` denotes all predecessors of `b` in the graph `BlockGraph`. For these `p` also the dominator relation to `b1` must hold. Note that `b` and `b1` are existentially quantified variables while `p` is all-quantified. The rule with predicate `SelfDom` is necessary because currently additions of single nodes to sets (in clauses) is not possible, everything has to be expressed in terms of edges (predicates).

OPTIMIX provides *functor transparency*, i.e. it is transparent which functors have been used to implement the graphs. This is automatically infered from the data model. The code for the graph navigations (functor method calls, access function calls) is generated accordingly.

The call sequence in a calling program could be:

```
ComputeDominators(BlockGraph, Dominators, SelfDom);
```

Note: The non-ground facts are computed first in each stratum.

5.2.1 Live Variables: MAY dataflow analysis

It is also possible to specify MAY data flow analysis. For that we need a bipartite graph functor (e.g. in CoSy-fSDL BIPUNI). It serves to represent the information which variables live at which basic block, here at which entry and exit of which block (LIVEIN, LIVEOUT). We also need the information per each basic block, which local variables have been used in a basic block (USED).

```
EARS LiveVariables()
{
   RANGE b <= LIVEOUT;
   RULES

   LIVEOUT(b,o) :- BlockGraph.succ(b,b1), LIVEIN(b1,o).
   LIVEIN(b,o)  :- USED(b,o).
   LIVEIN(b,o)  :- LIVEOUT(b,o).
}
```

A variable is live at the entry of a block, if it is used in the block, or if it is live at the exit of the block. A variable is live a the exit of the block, if it lives at the entry of a successor block.

5.2.2 BusyVariables: MUST dataflow analysis

If we want to solve a MUST data-flow analysis (intersection over all predecessors), we have to use an all quantifier. The following EARS computes busy local variables, e.g. variables that are used in all successor blocks or ar used in the block itself. The change is minimal.

```
EARS BusyVariables()
{
   RANGE b <= BUSYIN;
   RULES
   BUSYOUT(b,o) :- FORALL b1: BlockGraph.succ(b,b1), BUSYIN(b1,o).
   BUSYIN(b,o)  :- USED(b,o).
   BUSYIN(b,o)  :- BUSYOUT(b,o).
}
```

In similar fashion available expressions or busy expressions can be solved.

5.3 The generated code

5.3.1 Outline of the generated code

The outline of a generated file is as follows. There may be differences how the code is generated for one rule, whether a fixpoint evaluation is generated, etc.. Lines that are marked by a * appear as lists of items.

```
Macros for debugging and target code
Global target codes (BEGIN, IMPORT, ..)
Macro definitions for opaques (only CoSy-fSDL)
Routine *:
   Stratum *:
      Stratum variable definitions
         - order loop nodes
         - fixpoint check variables
      Evaluation of range declarations (get order loop node sets) *
      Fact evaluation *
      Index creation
      Fixpoint loop:
```

```
         Order loops *:
            Rule evaluations *:
               Assigning of the source variables of each rule from order loop nodes
               Rule code:
                  Rule variable definitions
                  FIRST code
                  Rule test:
                     Nested-loop join of all paths of edge-disjoint
                     path cover of the left-hand-side
                     ...
                     Evaluation  of join conditions
                     --------------
                     Transformation:
                        node allocations
                        edge deletions
                        edge additions
                        node marking/deletions
                     --------------
                     LAST code
Begin/Close procedures.
```

5.3.2 Manipulation and debugging of the generated code

We have tried to make the generated code as readable as possible. We hope users are able to read it and also make modifications. One can use OPTIMIX to get a skeleton for one's algorithm and then modify and refine it by hand. A lot of typing can be avoided in this respect.

RCS and SCCS ids are already generated, so that files directly can be imported under change control.

OPTIMIX generates some test code which is dependent on the flag `OXDEBUG`. If you set this manually in the code or set the `-DOXDEBUG` flag during a make, the running code will produce some test output. Users also can insert target predicates with print-statements and #ifdef-switches in order to print debug information.

However, the actual printing of the test output is dependent on the value of some option/variable of/in the engine. This is

- **-DUSE_SEQPAR_COSY** If this compilation switch is set (in CoSy-fSDL-mode), then a query in the option database of the engine is done for the string `"oxdebug"`. Thus, if the engine has got the option `"oxdebug"`, then test output is printed.

  In order to test the option, OPTIMIX generates a call to `engineStateGet`, which delivers the engine state. It is assumed to have a field `options`, which contains the engine option database. Thus the query is

  ```
  if (engineStateGet->options != NULL)
     /* test output */
  ```

  Note that users *must* save the options into the engine state at engine initialization.

- **-UUSE_SEQPAR_COSY** If this compilation switch is un-set then the global variable `int oxdebug;` is queried if output is to be printed. This is the normal case for AST-mode. In Cosy-fSDL-mode this is useful only if everything is clustered into one process.

  There is a second test print system which works in the same way. However, it prints less test output and is dependent on the engine option `"oxblip"`, or the global variable `int oxblip;`, respectively.

Output of graphs via VCG

All functors of the sol-library and all CoSy-fSDL-functors incooperate print routines which print a graph in VCG format to a file. You may call these routines in a target predicate to look at the current shape of a graph. You have to supply a function which provides a text label for each node. This function takes a node and returns a label string which is printed in the file. E.g. for a `hgraph` this works as follows:

```
{* extern char* labelfunction(<NodeType>);
   hgraph_print_vcgfile("example.grl",graph, labelfunction); *}
```

5.3.3 Miscellaneous

Opaque attribute types

For opaques in CoSy-fSDL (scalar attributes) OPTIMIX generates a set of macros which it uses to compare, assign and print. The user may redefine them.

Unknown types in the generated code

For navigations in the generated code OPTIMIX has to define some variables. In CoSy-fSDL-mode, sometimes their types are not known when the user compiles a generated file. This is the case for `SET_<NodeType>`, if the functor application `SET(<NodeType>)` does not appear in the CCMIR nor in view specifications. However, the generated file requires this type, because sets of `NodeType` are constructed during the navigations. The solution is that the user has to instruct fsdc to generate the domain with a CoSy-fSDL `use SET(<NodeType>)`-clause. Alternatively in some operator a dummy definition for `SET(mirBasicBlock)` can be introduced so that the fsdc generates this type as a result of this functor call.

Chapter 6

Practice

6.1 Implementation restrictions

The names of predicates (and corresponding object fields) must be globally unique within one GRS. This means that a predicate can be used only with one single type in a GRS. Field names may occur in different objects, however, then only one of them may be refered to in a GRS. Otherwise the typecheck-algorithm of OPTIMIX may deliver unexpected results.

It is currently not possible to define target code global variables for entire GRS, nor for a OPTIMIX-module. It is only possible for a single stratum. Thus passing variables around between strata is not easy. One trick is to include additional variables in the parameter list of the GRS. Then they are known globally.

Note that currently it is very simple to add multiple edges between the same nodes in EGRAPHs. Then the result of the generated routine may be unexpected.

Currently for non-recursive EARS no fixpoint evaluation is generated. This is only correct, if the computed predicates do not rely on each other. The rules are currently not sorted along their rule dependency graph. Be careful!

Note that the variables which are defined in patterns are not allowed to be used for further navigation, only for the use in target predicates, e.g. to test attributes. Also it is not allowed to use constant patterns. Constant comparisons are only allowed in equality tests (section 3.6.3).

Currently there is no automatic stratification of rules in strata.

The code generated for order loops may be buggy, because the implementation of the order algorithm is tricky.

6.1.1 Bidirectional edge optimization

Bidirectional edge optimization does only work for one-component rule test graphs, because currently we dont have a notion of an rule test graph with several components. For each of these a root must be found, if the component is cyclic!! Currently this is done only for the rule test graph completely. This results in a incorrect reduction of order because roots are forgotten.

6.1.2 Restrictions for modes

Several parts of the implementation have only been tested in AST- or CoSy-fSDL-mode. Thus inconsistencies are likely to appear. Please report them to the author.

6.2 Things to come

We plan to support and maintain OPTIMIX in the next years. The following things are probable for one of the next versions of OPTIMIX. Mail me if you need some different things.

1. Foreign functors. OPTIMIX should be able to deal with graph and set implementations which are external. This will be achieved by a macro interface.
2. Termination of XGRS should be automatically checked.
3. Stratification of XGRS.
4. The designator operators for attributes of objects: `V->attr`.

6.3 Frequently asked questions

Question: For a certain variable OPTIMIX infers 'type mismatched', i.e. multiple types. What can I do?

Answer: There are several reasons for this. Reason 1: OPTIMIX infers two domains that are compatible in CoSy-fSDL, but not in the flat form anymore, because in there the inheritance information of CoSy-fSDL is lost. Then state a FINER assertion that one domain is finer than the other and it should work.

Reason 2: There are really two different domains/types. Then help OPTIMIX by stating a type for that variable. You can do this either by a DECLARE variable declaration, which holds for all rules of a stratum. Or you can introduce pattern matching statements, whose type informations are then exploited for the type check. Or you qualify a predicate name by a domain/operator specification.

Reason 3: It really was a flaw in your specification. Look into the flatform-file, which types occur for your fields.

Question: For a certain variable OPTIMIX infers type `<NonIdentifiedClass>`

Answer: Probably in the specification there is a field whose type is not known, e.g. mis-typed. If the field has a functor call type, one of the parameters may be unknown. Look at the field name of the predicate in the line where the error occurs and look up its type in the data specification.

Question: My specification results in a larger order for my GRS than expected.

Answer: Maybe OPTIMIX has infered domains for the types of the root nodes of the rules which are different, however are compatible according to the domain calculus. Then insert a FINER statement at the beginning of the specification to tell OPTIMIX that two domains are compatible. OPTIMIX will then choose the coarser domain as type of the root node.

Answer: Maybe your FINER specification must be more detailed. Currently there is no union over different FINER specifications which contain the same tails. Be sure that you really specify all finer domains of a domain in one line.

Question: I try to compile the generated engine with `-DOXDEBUG`. However, it does not compile, because the field/variable `oxdebug` is unknown.

Answer (CoSy-fSDL): In order to use `OXDEBUG` you have to annotate the engine's state struct with a field `int state->oxdebug`. OPTIMIX-generated code then compares `engineStateGet->oxdebug` with the value of the given command-line option option. If the state does not have such a field, the engine does not compile. Also do not forget to save the value of the command-line option `oxdebug` in the state.

Answer (AST/standalone): Please supply in your main program two variables

```
int oxdebug;
int oxblip;
```

which are flags to guide the printing of the debugging output macros.

Question: I would like to specify an empty rule test in order to perform the rule's transformation always. However, OPTIMIX does not accept empty rule tests.

Answer 1: Try a non-ground fact. You may combine the with a target code, if you want. Currently facts consisting only of target code do not work. The disadvantage is that facts are always moved to the top of the generated code of a stratum.

Answer 2: Specify standalone matches on variables. The matches are generated, even if the variable's type is clear. Thus the rule test is not empty and the transformation will be generated also:

```
EARS ComputeTrafo(loop:Loop,MaxInstruction:Instruction)
{  RULES
   (
      MaxInstruction ~ STMT,
      loop           ~ Loop
   ==>
      new_instructions.last(loop,MaxInstruction)
   )
}
```

Question: I'm totally worried about this tool. How can I understand the weird documentation?

Answer: Just relax and try `optimix -poem`. You are not the only one who is perplexed by graph rewriting.

Glossary

edge-disjoint path cover A covering of a graph with a set of paths, which intersect each other only at their end points.

exhaustive graph rewrite system (XGRS) A variant of graph rewrite systems on relational graphs, i.e. on graphs with one edge of a certain label between two nodes. Each rule adds an edge but no node to a *termination subgraph*, thus the systems terminate

order A characteristic feature of a stratum or a GRS: the maximal number of root nodes of left hand sides.

order loop A loop in the generated code which traverses a domain of a root node of some left hand sides.

order loop node A node instantiated in an order loop.

nested-loop join A code generation method from DATALOG and relational algebra to evaluate relational queries.

node A variable which denotes a node of the rule test graph

pattern variable A variable set in a pattern. May be a node or a scalar variable.

rule test A left hand side of a graph rewrite system and a rule body of a DATALOG-query.

rule test graph, RTG A left hand side of a graph rewrite system consists of a graph, the rule test graph.

rule transformation A right hand side of a graph rewrite system.

scalar variable A variable with scalar value. No rule test graph node.

Bibliography

[Aß94] Aßmann, Uwe. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *5th Int. Workshop on Graph Grammars and Their Application To Computer Science, Williamsburg*, Lecture Notes in Computer Science, Spinger Verlag, pages 321–335, Heidelberg, November 1994. Springer.

[Aß95] Uwe Aßmann. *Generierung von Programmoptimierungen mit Graphersetzungssystemen.* PhD thesis, Universität Karlsruhe, Kaiserstr. 12, 76128 Karlsruhe, Germany, July 1995. GMD-Bericht.

[Aß96a] Uwe Aßmann. Graph Rewrite Systems For Program Optimization. unpublished draft of journal paper on exhaustive graph rewrite systems, 1996.

[Aß96b] Uwe Aßmann. How To Uniformly Specify Program Analysis and Transformation. In P. A. Fritzson, editor, *Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science, Spinger Verlag*, Heidelberg, 1996. Springer.

[ASU72] A. V. Aho, R. Sethi, and J. D. Ullman. Code Optimization And Finite Church-Rosser Systems. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 89–105. Prentice-Hall, 1972.

[Buh95] Claus-Thomas Buhl. fSDL Language Report. Technical report, COMPARE Consortium, 1995.

[CGT89a] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer, Heidelberg, 1989.

[CGT89b] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge And Data Engineering*, 1(1):146–166, March 1989.

[GE90] Josef Grosch and Helmut Emmelmann. A tool box for compiler construction. In *LNCS 477: Compiler Compilers; Third International Workshop, CC'90; Schwerin, FRG; Proceedings.* Springer, October 1990.

[Gro89] Josef Grosch. Ast - a generator for abstract syntax trees. Technical report, August 1989.

[Nag79] M. Nagl. *Graph-Grammatiken, Theorie, Implementierung, Anwendungen.* Vieweg, 1979.

[RSS92] R. Ramakrishnan, D Srivastava, and S. Sudarshan. CORAL - Control, Relations and Logic. In *Proceedings of the 18th VLDB Conference*, 1992.

[Tar81] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, July 1981.

Generated Example

```
/*  @(#)example-reachable.c     1.1 96/06/26 */
/* ========================================================== */
/*                                                            */
/*          GENERATED CODE. MODIFICATIONS WILL BE LOST        */
/*                                                            */
/* ========================================================== */
/*** generated by optimix  at Wed Jun 26 13:11:34 1996
Version: 1.1 linked at Date: Wed Jun 26 13:06:42 MET DST 1996
   source : example-reachable.ox
   view : example-reachable
   LocalTests : ON
   OptimizerOutputFileTrunc : example-reachable
   output : example-reachable
 ***/


/* Module TransitiveClosure                                  */
/* Debug macro. Switch on if you want test output. */
/* #define OXDEBUG */
/* for the next macro the engine state has to include an option component!! */
#ifdef OXDEBUG
#ifndef USE_SEQPAR_COSY
#define OXDEBUGBEGIN(number) {{ extern int oxdebug; if (oxdebug>number) {
#define OXDEBUGEND  }}}
#else
#define OXDEBUGBEGIN(number) { if (engineStateGet->options != NULL) { if (engineStateGet->oxdebug > (number)) {
#define OXDEBUGEND  }}}
#endif
#endif
#ifdef OXDEBUG
#ifndef USE_SEQPAR_COSY
#define OXBLIPBEGIN(number) {{ extern int oxblip; if (oxblip>number) {
#define OXBLIPEND  }}}
#else
#define OXBLIPBEGIN(number) { if (engineStateGet->options != NULL) { if (engineStateGet->oxdebug > (number)) {
#define OXBLIPEND  }}}
#endif
#endif
#ifdef OXDEBUG
#ifndef USE_SEQPAR_COSY
#define OXOPTBEGIN(Option) { if (options_IsSet(Option)) {
#define OXOPTEND  }}
#else
#define OXOPTBEGIN(Option) { if (optionIsSet(engineStateGet->options,Option)) {
#define OXOPTEND  }}
#endif
#endif
#define WRITENL fprintf(stderr,"\n");
/* includes ------------------------------------------------- */
# include "example-reachable.h"
# include <stdio.h>

#define TEST(x)    if( !(x) ) continue;
#define ABSENT(x)  if( (x) ) continue;

static void oxAbort (char *yyFunction)
{
 extern void exit ();
 (void) fprintf (stderr, "Error: module example-reachable, function %s failed", yyFunction);
 exit (1);
}

static char rcsid[] = "$Id:$";

/* line 0 */

/* macro definitions for opaques ------------------------------ */

/* help and debug routines ---------------------------------- */

/* Implementation of GRS ---------------------------------- */
/* ------------------------------------------------------------ */
/* It is assumed that several parameter graphs are given (in    */
/* ascending alphabetical order).                               */
/* Some of them are assigned, others are only used for test.    */
/* Note that the universes of the graphs concerning their node types */
/* MUST be the same.                                            */
/* Nodes in the assigned graphs must be already in the graphs;  */
/* this routine only computes the edges.                        */
/* TEST THIS CODE BEFORE YOU TRUST IT!                          */
/* ComputeReachableBlocks is a EARS(0).                         */
/* ------------------------------------------------------------ */
void ComputeReachableBlocks (consset BlockSet)
{
int _change, _fixcount;
```

```
#ifdef OXDEBUG
OXDEBUGBEGIN(0)
fprintf(stderr,"ComputeReachableBlocks"); WRITENL;
OXDEBUGEND
#endif /* OXDEBUG */
/* stratum code generation ======================================== */
/* ---------------------------------------------------------------- */
/* stratum-18:6 is a recursive stratum with order 1.                */
{
        int _OLoopCounters[1];
        short _order_onceindex;
        unsigned char AllquantorFailFlag = (unsigned char)0;
        /* order loop 0: set or graph range */
        Block OLoopNodeRepr_0;
        Block OLoopRawNode_0;
        consset OLoopNodeSet_0;
        /* Indices */
        /* Global: result sets */
#ifdef OXDEBUG
OXDEBUGBEGIN(0)
fprintf(stderr,"stratum-18:6"); WRITENL;
OXDEBUGEND
#endif /* OXDEBUG */
        /* Get the order loop nodes from some tested graphs     */
        OLoopNodeSet_0 = BlockSet;

        /* Non-ground fact evaluation ========================= */

        /* Index creation  ==================================== */
        /* Rule evaluation ==================================== */
        _change = TRUE;
        _fixcount = 0;
        while (_change)
        {
        _change = FALSE;
#ifdef OXDEBUG
OXDEBUGBEGIN(0)
fprintf(stderr,"enter fixpoint loop %d",_fixcount);WRITENL;
OXDEBUGEND
#endif /* OXDEBUG */
#ifdef OXDEBUG
OXBLIPBEGIN(4)
fprintf(stderr,"enter fixpoint loop %d",_fixcount);WRITENL;
OXBLIPEND
#endif /* OXDEBUG */
        _fixcount++;
        /* rules with order loops ============================= */
        _OLoopCounters[0] = 0;
        consset_LOOP(OLoopNodeSet_0,OLoopNodeRepr_0)
        {
#ifdef OXDEBUG
OXBLIPBEGIN(4)
fprintf(stderr,"order loop 0 run %d ", _OLoopCounters[0]++);WRITENL;
OXBLIPEND
#endif /* OXDEBUG */
                OLoopRawNode_0 = OLoopNodeRepr_0;

                /* ready rule tests ========================= */
                /* edge addition rule test   23,  4 ------------------------------ */
#ifdef OXDEBUG
OXDEBUGBEGIN(0)
                fprintf(stderr,"rule test   23,  4 "); WRITENL;
OXDEBUGEND
#endif /* OXDEBUG */
                {
                        short _onceindex;
                        /* raw nodes */
                        Block b;
                        Block b1;

                        /* for edge with source node */
/* Nothing to be done for edge source of set edge */
                        /* for tested edge BlockGraph */
                        consset RuleNodeCursor_BlockGraph_b1;
                        consset RuleNodeSet_BlockGraph_b1;

                        /* pattern variables */
                        /* Local: result sets */

                        /* rule fake for-loop, executed only once.. */
                        for (_onceindex = 1; _onceindex > 0; _onceindex--)
                        {
                                /* path test [b]-BlockGraph->[b1] */
                                b = OLoopRawNode_0;
                                /* predicate test (set edge) BlockGraph  23, 29 */
#ifdef OXDEBUG
OXDEBUGBEGIN(0)
fprintf(stderr,"predicate   23, 29 order loop node %s ", Tree_get_Label(b)); WRITENL;
OXDEBUGEND
#endif /* OXDEBUG */
                                RuleNodeSet_BlockGraph_b1 = b->BlockGraph;
                                consset_LOOP(RuleNodeSet_BlockGraph_b1, b1)
                                {
                                        /* evaluate first target predicates */
                                        /* global attribute tests (join mode) */
                                        /* evaluate join conditions */
                                        /* assign all edges between tested nodes */
                                        /* assign edge ReachableBlocks   23,  4 */
                                        _change |= consset_insert(b->ReachableBlocks,b1);
                                }
                                consset_ENDLOOP;
                                /* end path test [b]-BlockGraph->[b1] */
                                /* Rule finish */
                        } /* end of rule fake for-loop */
FAILRULE23:
                } /* end of rule test   23,  4 */
                /* edge addition rule test   24,  4 ------------------------------ */
#ifdef OXDEBUG
OXDEBUGBEGIN(0)
```

```
                fprintf(stderr,"rule test   24,  4 "); WRITENL;
OXDEBUGEND
#endif /* OXDEBUG */
                {
                        short _onceindex;
                        /* raw nodes */
                        Block b;
                        Block b1;
                        Block s;

                        /* for edge with source node */
/* Nothing to be done for edge source of set edge */
                        /* for tested edge BlockGraph */
                        consset RuleNodeCursor_BlockGraph_s;
                        consset RuleNodeSet_BlockGraph_s;

                        /* for tested edge ReachableBlocks */
                        consset RuleNodeCursor_ReachableBlocks_b1;
                        consset RuleNodeSet_ReachableBlocks_b1;

                        /* pattern variables */
                        /* Local: result sets */

                        /* rule fake for-loop, executed only once.. */
                        for (_onceindex = 1; _onceindex > 0; _onceindex--)
                        {
                                /* path test [b]-BlockGraph->[s]-ReachableBlocks->[b1] */
                                b = OLoopRawNode_0;
                                /* predicate test (set edge) BlockGraph  24, 29 */
#ifdef OXDEBUG
OXDEBUGBEGIN(0)
fprintf(stderr,"predicate   24, 29 order loop node %s ", Tree_get_Label(b)); WRITENL;
OXDEBUGEND
#endif /* OXDEBUG */
                                RuleNodeSet_BlockGraph_s = b->BlockGraph;
                                consset_LOOP(RuleNodeSet_BlockGraph_s, s)
                                {
                                        /* predicate test (set edge) ReachableBlocks  24, 46 */
                                        RuleNodeSet_ReachableBlocks_b1 = s->ReachableBlocks;
                                        consset_LOOP(RuleNodeSet_ReachableBlocks_b1, b1)
                                        {
                                                /* evaluate first target predicates */
                                                /* global attribute tests (join mode) */
                                                /* evaluate join conditions */
                                                /* assign all edges between tested nodes */
                                                /* assign edge ReachableBlocks   24,  4 */
                                                _change |= consset_insert(b->ReachableBlocks,b1);
                                        }
                                        consset_ENDLOOP;
                                }
                                consset_ENDLOOP;
                                /* end path test [b]-BlockGraph->[s]-ReachableBlocks->[b1] */
                                /* Rule finish */
                        } /* end of rule fake for-loop */
FAILRULE24:
                } /* end of rule test   24,  4 */

        } /* end of order loop 0 */
        consset_ENDLOOP;
        /* single source path problems ======================== */
        }               /* end of fix point loop */

FAILSTRATUM_18_6:
}               /* end of stratum stratum-18:6 */


}               /* end of ComputeReachableBlocks */


/* Initialising and finishing ---------------------------------- */
/* line 0 */
void example-reachable_Begin ()
{
}

/* line 0 */
void example-reachable_Close ()
{
}
/*********************

     Unendlich staun ich euch an, ihr Seligen, euer Benehmen,
     wie ihr die schwindliche Zier traget in ewigem Sinn.
     Ach wer's verstuende zu bluehn: dem waer das Herz ueber alle
     schwachen Gefahren hinaus und in der grossen getrost.
                  R.M. Rilke: Mandelbaeume in Bluete

*********************/
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l' Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399